Software and System Architecture Evaluation and Analysis in the Automotive Domain

Von der Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik
der Technischen Universität Carolo-Wilhelmina
zu Braunschweig

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

genehmigte

Dissertation

von

M.Sc. Dipl.-Inform. Bastian Florentz

geboren am 08.10.1977

in Duisburg

Eingereicht am:	25.10.2007
Mündliche Prüfung am:	07.02.2008
Referentin/Referent:	Prof. Dr. rer. nat. Ursula Goltz
Korreferentin/Korreferent:	Prof. Dr.-Ing. Stefan Kowalewski

2008

Acknowledgements

This thesis has been prepared during my time at the Technische Universität Braunschweig at the institute of Ursula Goltz. I would like to thank Ursula for giving me the opportunity of working in her group and supporting me in many respects. In particular, she sent me to conferences and provided the opportunity of participating in industrial projects.

My colleagues at Ursula's institute deserve greate thanks as well:

Martin Mutz, who introduced me into scientific working by supervising my master thesis and attracted my interest for staying at the university.

Michaela Huhn, who supervised me in scientific writing and as coauthor of a good share of my publications. She taught me in preparing and presenting my work in the scientific community.

In particular for this thesis, I am indepted to Stefan Kowalewski, the second reviewer, as well as Rolf Ernst and Ulrich Golze as members of the examination commission. I would like to thank Werner Struckmann and Malte Lochau for reams of useful hints in preparing my thesis. Great parts of one of the case studies of my thesis have been prepared in close collaboration with Torben Mielke, one of my students. Thank you, Torben, for your engagement, endurance, and so many productive discussions.

Many thanks to all my other colleaques for an enjoyable atmosphere and many discussions during my time at the institute.

Finally, heartfelt thanks to my family, especially my parents, my grandma—in grateful remembrance—, my brother, and my friends for believing in me, and Astrid for her patience with me while preparing my thesis. Thank you all.

Abstract

The complexity of software-intensive embedded systems in the automotive domain has been steadily growing in the recent years. As a consequence, controller-oriented development processes will be replaced by function-oriented ones. The main intention of function-oriented development is an initial decoupling of hardware and software. This increases the flexibility during development. Even more, the reuse of hardware and software components is supported. To handle these capabilities, a superior view on the structure of a system and its development rationale becomes necessary which is called the architecture of the system. Architecture evaluation has to be performed to assess the suitability of an architecture with respect to extra-functional requirements on the system, i.e. quality attributes. Thus, the development rationale can be expressed according to the quality attributes.

In this work, an evaluation structure is proposed which is called the Quality Attribute Directed Acyclic Graph. It provides an explicit representation of extra-functional requirements and will be the basis not only for evaluation but also for architecture analysis. An evaluation methodology in terms of evaluation tactics for time-saving and cost-efficient evaluation processing is defined. Thus, an often great number of possible architecture variants can be handled without causing too much evaluation and thus development effort.

Architecture analysis will be performed to understand evaluation results and to reason on development rationales. Analysis results are a valuable feedback for current as well as for future development projects. Architectural decisions on the system composition can be supported as their impact on the evaluation result can be predicted. Furthermore, promising architectural changes of architecture variants can be identified which again saves development effort.

With growing and moreover expressible architectural knowledge, decisions can be supported more efficiently and even guided architecture development becomes conceivable. Besides documentation and communication, the whole development process profits from decision support as even quickly made decisions become more reliable.

Zusammenfassung

Die Komplexität Software-intensiver eingebetteter Systeme im Automobil ist in den letzten Jahrzehnten ständig gestiegen. Daher werden Steuergeräte-orientierte Prozesse durch Funktionen-orientierte ersetzt. Die Motivation Funktionen-orientierter Entwicklung liegt in der Entkopplung von Hardware und Software, um die Flexibilität in der Entwicklung zu erhöhen und die Wiederverwendung von Hardware und Software zu unterstützen. Um diese Möglichkeiten handhaben zu können, bedarf es einer übergeordneten Sicht auf das System und die angewendeten Prinzipien in der Entwicklung. Diese Sicht wird als Architektur des Systems bezeichnet. Durch Architekturevaluation wird die Eignung einer Architektur bezüglich extra-funktionaler Anforderungen an das System, so genannter Qualitätsattribute, festgestellt. Die angewendeten Entwicklungsprinzipien können vor dem Hintergrund der Qualitätsattribute aufgezeigt werden.

In dieser Arbeit wird eine Evaluationsstruktur, der Quality Attribute Directed Acyclic Graph, vorgeschlagen. Sie bietet eine explizite Darstellung der extra-funktionalen Anforderungen und ist nicht nur die Grundlage für die Evaluation, sondern auch für Architekturanalyse. Eine Evaluationsmethodik wird in Form von Taktiken definiert, die eine zeitsparende und kosteneffiziente Durchführung der Evaluation versprechen. Daher kann eine oftmals sehr große Anzahl möglicher Architekturvarianten berücksichtigt werden, ohne dass zu großer Evaluations- und somit Entwicklungsaufwand entsteht.

Architekturanalyse wird eingesetzt, um Evaluationsergebnisse besser zu verstehen und angewendete Entwicklungsprinzipien zu diskutieren. Die Ergebnisse der Architekturanalyse stellen eine wertvolle Rückmeldung in Bezug auf die erreichte Qualität sowohl in aktuellen als auch zukünftigen Projekten dar. Architekturentscheidungen bezüglich der Systemkomposition werden unterstützt, da der entsprechende Einfluss auf die Evaluationsergebnisse im Vorfeld abgeschätzt werden kann. Des Weiteren können vielversprechende Veränderungen von Architekturvarianten identifiziert werden, was abermals Entwicklungsaufwand spart.

Mit steigendem und vor allem auch formulierbarem Architekturwissen können Architekturentscheidungen immer effizienter unterstützt werden. Sogar Architekturentwicklung mit entsprechender Werkzeug-basierter Unterstützung rückt in greifbare Nähe. Neben Dokumentation und Kommunikation profitiert der gesamte Entwicklungsprozess von einer Entscheidungsunterstützung, die schnelle und zuverlässige Entscheidungen ermöglicht.

Contents

List of Tables xiii

List of Figures xv

1 Introduction 1
- 1.1 Architecture in Software and System Development . . . 2
- 1.2 Software versus System Architecture . . . 5
- 1.3 Architectural Decisions . . . 6
  - 1.3.1 Decision Impact . . . 8
  - 1.3.2 Predictability . . . 9
- 1.4 Objectives . . . 10
- 1.5 Outline . . . 11

2 Embedded System Architecture Views 13
- 2.1 Architecture Modeling . . . 14
  - 2.1.1 Types, Instances, and Refinement . . . 14
  - 2.1.2 Viewtypes, Styles, and Views . . . 14
  - 2.1.3 Components and Connectors . . . 15
  - 2.1.4 System and Environment . . . 15
  - 2.1.5 Functionality, Functions, and Features . . . 16
  - 2.1.6 Architectural Decisions and Variants . . . 18
- 2.2 Component-and-Connector Viewtype . . . 18
- 2.3 The Function Architecture . . . 20
  - 2.3.1 Functions . . . 21
  - 2.3.2 Signals . . . 22
  - 2.3.3 Signal-to-signal Connector . . . 23
  - 2.3.4 Types and Instances . . . 24
- 2.4 The Hardware Architecture . . . 25
  - 2.4.1 Controllers . . . 26
  - 2.4.2 Sensors and Actuators . . . 28
  - 2.4.3 Communication Lines . . . 29
  - 2.4.4 Device-to-commline Connector . . . 29
  - 2.4.5 Omission of Power Supply . . . 30
  - 2.4.6 Network Layout . . . 31

2.4.7 Types and Instances . . . 32
2.5 The Function Mapping . . . 33
2.5.1 The Components . . . 34
2.5.2 Function-to-controller Connector . . . 34
2.6 The Communication Mapping . . . 35
2.6.1 The Components . . . 35
2.6.2 Signal-to-commline Connector . . . 36
2.6.3 Subsystem Interface . . . 40
2.6.4 Communication Mapping Alternatives . . . 41
2.7 Architecture View Graphs . . . 42

3 Architecture Case Studies 45
3.1 Body Comfort System Architecture . . . 45
3.1.1 Legacy Power Window Control System . . . 46
3.1.2 Short Lift Control . . . 48
3.1.3 Convertible Top Control . . . 50
3.1.4 Battery Capacity . . . 53
3.1.5 The Variants . . . 55
3.2 In-Car Radio Navigation System Architecture . . . 56
3.2.1 The Variants . . . 56

4 Architecture Evaluation 57
4.1 Architecture Quality . . . 57
4.2 The Quality Attribute Directed Acyclic Graph . . . 61
4.2.1 Quality Attribute Hierarchy . . . 62
4.2.2 Evaluation Techniques . . . 63
4.2.3 Joining Techniques . . . 65
4.2.4 Interpretation of Results . . . 65
4.2.5 Scenarios . . . 67
4.2.6 Constraints . . . 69
4.3 Constructing an Evaluation . . . 70
4.3.1 Hierarchical Composition . . . 70
4.3.2 Quality Attribute Result Joining . . . 72
4.3.3 Setting up Quality Attributes . . . 75
4.4 Related Work – Architecture Evaluation . . . 76

5 Evaluation Processing Methodology 83
5.1 Dimensions of Architecture Evaluation . . . 84
5.1.1 Evaluation Sensitivity . . . 84
5.1.2 Evaluation Effort . . . 88
5.1.3 Evaluation Soundness . . . 91
5.2 Evaluation Tactics . . . 94

5.2.1 Great Structural Impact . . . 96
5.2.2 K.O. Attributes . . . 96
5.2.3 Available Input . . . 97
5.2.4 Easy Setup . . . 98
5.2.5 Easy Processing . . . 98
5.2.6 Scope Synergy . . . 99
5.2.7 Why Soundness Does Not Contribute . . . 100
5.3 Related Work – Evaluation Methods . . . 100

6 Evaluation Case Studies **105**
6.1 Body Comfort System Evaluation . . . 105
6.1.1 Communication Performance . . . 106
6.1.2 CPU Performance . . . 108
6.1.3 RAM Performance . . . 110
6.1.4 ROM Performance . . . 111
6.1.5 Costs . . . 113
6.1.6 Weight . . . 114
6.1.7 Battery Standby Time . . . 116
6.1.8 Battery Life Time . . . 117
6.1.9 Extendability . . . 119
6.1.10 Scalability . . . 120
6.1.11 Evaluation Results . . . 120
6.2 In-Car Radio Navigation System Evaluation . . . 123
6.2.1 Costs . . . 123
6.2.2 Performance . . . 124
6.2.3 Modifiability . . . 127
6.2.4 Evaluation Results . . . 127

7 Architecture Analysis **131**
7.1 Architecture Sensitivity . . . 131
7.1.1 Dependency and Sensitivity . . . 132
7.1.2 Single Objective Sensitivity . . . 133
7.1.3 Multiple Objective Sensitivity . . . 135
7.2 The Modular Performance Analysis . . . 137
7.2.1 Real-Time Calculus . . . 137
7.2.2 Timed Automata . . . 138
7.2.3 Modular Performance Analysis Model . . . 139
7.2.4 Modular Performance Analysis Results . . . 141
7.3 Architecture Potential Analysis . . . 142
7.3.1 Identifying Relevant Sensitivities . . . 142
7.3.2 Quality Correlation . . . 145
7.4 Related Work – Architecture Analysis . . . 147

7.4.1 The SymTA/S Approach 150

8 Analysis Case Studies **153**
8.1 Body Comfort System Analysis 153
8.1.1 Body Comfort System Result Screening 154
8.1.2 Body Comfort System Insights 157
8.2 In-Car Radio Navigation System Analysis 159
8.2.1 In-Car Radio Navigation System Tradeoff Analysis 159
8.2.2 In-Car Radio Navigation System Insights 163

9 Conclusion and Outlook **165**
9.1 Conclusion 165
9.2 Adaptability 166
9.3 Outlook 167

Bibliography **171**

A BCS Component Properties **181**

B BCS Evaluation Results **189**

List of Tables

2.1 The function architecture 21
2.2 The hardware architecture 26
2.3 The function mapping 33
2.4 The communication mapping 35

3.1 BCS variants overview 55

4.1 Topmost quality attributes of the Body Comfort System case study . 71
4.2 Physics refinement in the Body Comfort System case study 71
4.3 Costs refinement in the Body Comfort System case study 72
4.4 Analytic Hierarchy Process for the BCS case study 73
4.5 Weights and impact of the BCS case study 74

6.1 CPU performance of the Body Control Device (8 MIPS) 109
6.2 RAM utilization of the Body Control Device (8 kbyte) 110
6.3 ROM utilization of the Body Control Device (256 kbyte) 112
6.4 Battery standby time of the Body Comfort System 116
6.5 Battery life time of the Body Comfort System 118
6.6 BCS evaluation results overview 121
6.7 Evaluation results Variant SLC on PWD, CTC on MiniCTD, 61 Ah . 121
6.8 Evaluation results Variant SLC on PWD, CTC on BCD, 61 Ah 122
6.9 Evaluation results Variant SLC on BCD, CTC on BCD, 61 Ah 122
6.10 Evaluation results Variant SLC on BCD, CTC on BCD, 50 Ah 122
6.11 Evaluation results Variant I 128
6.12 Evaluation results Variant II 128
6.13 Evaluation results Variant III 129

7.1 Order patterns for Decisions B and C 143

8.1 BCS evaluation result patterns 155
8.2 BCS evaluation result groups 156
8.3 Results of the 2nd, 3rd, 6th, and 7th variants in the ranking 157
8.4 ICRNS weights and impact 160

A.1 BCS signal properties 181

A.2 BCS properties of other software 181
A.3 BCS function properties 182
A.4 BCS sensor and actuator properties 183
A.5 BCS ECU properties 183
A.6 BCS cable properties 184
A.7 BCS battery properties 184
A.8 BCS LIN messages for 5 ms period schedule 185
A.9 BCS LIN messages for 10 ms period schedule 186
A.10 BCS CAN messages 187

B.1 Evaluation results Variant SLC on PWD, CTC on MiniCTD, 72 Ah . 189
B.2 Evaluation results Variant SLC on PWD, CTC on BCD, 72 Ah 189
B.3 Evaluation results Variant SLC on BCD, CTC on MiniCTD, 61 Ah . 190
B.4 Evaluation results Variant SLC on BCD, CTC on MiniCTD, 72 Ah . 190
B.5 Evaluation results Variant SLC on BCD, CTC on BCD, 72 Ah 190
B.6 Evaluation results Variant SLC on BCD, CTC on MiniCTD, 50 Ah . 191
B.7 Evaluation results Variant SLC on PWD, CTC on MiniCTD, 50 Ah . 191
B.8 Evaluation results Variant SLC on PWD, CTC on BCD, 50 Ah 191
B.9 Evaluation results Variant SLC on PWD, CTC on RedCTD, 72 Ah . 192
B.10 Evaluation results Variant SLC on PWD, CTC on RedCTD, 61 Ah . 192
B.11 Evaluation results Variant SLC on PWD, CTC on RedCTD, 50 Ah . 192
B.12 Evaluation results Variant SLC on BCD, CTC on RedCTD, 72 Ah . . 193
B.13 Evaluation results Variant SLC on BCD, CTC on RedCTD, 61 Ah . . 193
B.14 Evaluation results Variant SLC on BCD, CTC on RedCTD, 50 Ah . . 193

List of Figures

1.1 Decision levels 7

2.1 Viewtypes, styles, and views, cf. [CBB+02] 15
2.2 The component-and-connector viewtype 19
2.3 Scheme of views defined for the embedded system architecture 20
2.4 The function architecture metamodel 22
2.5 The hardware architecture metamodel 27
2.6 The components of the hardware architecture 27
2.7 PREEvision topology model 32
2.8 The function mapping metamodel 34
2.9 The communication mapping metamodel 36
2.10 The elements of the views mapped to OSI reference model 37
2.11 View graph legend 42
2.12 Connections between controller and sensors/actuators 42
2.13 Connections to communication lines 43
2.14 Three styles of function mapping visualization 43
2.15 Alternative architecture view graph 44

3.1 Legacy Power Window Control System 47
3.2 SLC on Body Control Device 48
3.3 SLC on Power Window Devices 49
3.4 CTC on Body Control Device 51
3.5 CTC on Reduced Convertible Top Device 51
3.6 CTC on Mini Convertible Top Device 52
3.7 Overview of the 18 BCS variants 54
3.8 Four view graphs representing 12 variants 54
3.9 Variants of the In-Car Radio Navigation System 56

4.1 QADAG metamodel 62
4.2 Quality attribute hierarchy (part of the metamodel) 62
4.3 Example QADAG layouts 63
4.4 Interpretation of results 66
4.5 Scenario items by [BCK98] 68
4.6 Address lookup scenario, cf. [WTVL06] 68
4.7 Address lookup scenario presentation according to [BCK98] 68

4.8 Hierarchy and impact . . . 75
4.9 ISO/IEC 9126 quality model for external and internal quality . . . 77
4.10 ISO/IEC 9126 quality in use model . . . 77
4.11 Sample utility tree, cf. [KKC00] . . . 78
4.12 Evaluation matrix, cf. [BRST05] . . . 80

5.1 Separator between structural and architectural impact in a QADAG 85
5.2 Weights and impact in a QADAG . . . 85
5.3 Interpretation-based impact regarding costs . . . 87
5.4 Tradeoff between hardware costs and performance . . . 87

6.1 BCS case study QADAG . . . 106
6.2 BCS communication utilization interpretation . . . 107
6.3 BCS CPU utilization interpretation . . . 109
6.4 BCS RAM utilization interpretation . . . 111
6.5 BCS ROM utilization interpretation . . . 112
6.6 BCS costs interpretation . . . 114
6.7 BCS weight interpretation . . . 115
6.8 BCS battery standby time interpretation . . . 117
6.9 BCS battery life time interpretation . . . 118
6.10 ICRNS case study QADAG . . . 123
6.11 ICRNS costs interpretation . . . 124
6.12 ICRNS expected delay interpretation . . . 125
6.13 Change volume scenario, cf. [WTVL06] . . . 125
6.14 Maximal end to end delays, see [WTVL06] . . . 126

7.1 Dependencies in architecture development . . . 132
7.2 Dependencies regarding single objectives . . . 133
7.3 Dependencies regarding multiple objectives . . . 135
7.4 MPA event processing, see [WTVL06] . . . 137
7.5 Radio function automaton, see [HV06] . . . 138
7.6 Bus automaton, see [HV06] . . . 139
7.7 TMC message handling scenario, cf. [WTVL06] . . . 140
7.8 MPA model, see [WTVL06] . . . 140
7.9 MPA result (Variant II), see [WTVL06] . . . 141
7.10 TMC delay versus MMI processor speed, cf. [WTVL06], Fig. 18 . . . 145
7.11 Correlation examples . . . 146
7.12 Correlation chain over three domains . . . 146
7.13 Correlation of quality interpretations . . . 147
7.14 Context for the CBAM, cf. [KAK01, KAK02] . . . 148
7.15 Utility response curves examples, cf. [KAK02] . . . 149

8.1 ICRNS interpretations to be correlated . . . 159

8.2 MPA results of MMI/RAD processor speed (Variant II) 160
8.3 Costs dependency on processor speed 161
8.4 Costs-delay dependencies, left: without network, right: with network 161
8.5 Correlation of hardware costs and (expected) delay 162
8.6 Architecture potential without network (Variant III) 163
8.7 Architecture potential with network (Variant II) 163

1 Introduction

Software-intensive embedded systems have found their way into many every day life products. The automotive domain is only one example for an extensive integration of such embedded systems. A good share of the product functionality is controlled and realized by an embedded system and growing functionality means a growing embedded system as well. To handle the complexity of such systems, model-based development processes have been established (see Florentz et al. [FMH04, FH04, FHB06], Huhn et al. [HMF04, HMD+04], Rau [Rau02], Schätz et al. [SPHP02], and others). Furthermore, the complexity requires a structured view on the software and system containing the realized design principles, i.e. the architecture rationale, (see Perry and Wolf [PW92], Kruchten [Kru95], and Clements et al. [CBB+02]). This view is called the software and system architecture. Architecture development has become an essential part of the software and system development (see Shaw and Garlan [SG96, GS93, GS94], Hofmeister et al. [HNS00], Reussner and Hasselbring [RH06], and Gorton [Gor06]).

In the automotive domain, in which system development is going to change from controller-oriented processes to function-oriented ones according to the AUTomotive Open System ARchitecture approach (AUTOSAR, see Heinecke et al. [HSF+04]), architecture will be in the center of interest. Function-oriented development decouples software and hardware to increase the flexibility of development. The main objective is to reduce the complexity of the embedded systems based on the integration of up to eighty controllers and several sensors and actuators in a single system. Therefore, fundamental decisions on the structure of the system have to be made early in the development process, i.e. during architecture development. After all, the initial decoupling of hardware and software later requires a thoughtful distribution of software components on the available hardware and is no guaranty for successful development on its own. So-called architectural decisions have to be made carefully and are the first to be made in the development and thus significantly affect the quality of the system under development.

To assess the quality of architecture, evaluation has to be performed and documented to identify the most promising architecture variants and to profit from experiences made during development. Architecture evaluation is directed to extra-functional requirements, so-called quality attributes, which state how well a systems realizes its functionality. As such requirements cover different concerns like performance, reliability, modifiability, costs, and so on, a structured representation of the quality attributes is necessary. Furthermore, the meaning of an architecture quality

requirement needs to be expressed in order to provide a good basis for evaluation. In this approach, the Quality Attribute Directed Acyclic Graph will be introduced. It will be used to model architecture quality requirements in detail and thus is the backbone of architecture evaluation and analysis. The results of architecture evaluation and analysis provide the basis for understanding architecture rationale even in case of being a non-expert regarding some of the quality attributes contained in the evaluation.

While high quality can be attributed to the system architecture itself, low development effort is mainly based on decision support. The explicit architecture quality requirement representation by the Quality Attribute Directed Acyclic Graph supports decisions with respect to the identification of promising architecture variants and in terms of architectural changes. Evaluation processing methodology for efficient evaluation is mainly based on the quality attributes, i.e. the respective evaluation techniques. Analysis for architecture potential takes detailed dependencies between architecture and its evaluation into account. Hence, tradeoffs between quality attributes can be investigated to identify promising changes of architecture variants. With respect to design space exploration as an established development discipline, analysis for promising changes can be considered as architecture space exploration. Evaluation as well as analysis results can be fed back, which increases decision support in current and in future development projects. Especially domains of software-intensive high-quality systems with short innovation cycles can highly profit from time-saving and cost-efficient system development. Decision support for efficient evaluation and architecture space exploration is the first step to integrate architecture development and evaluation approaches for improving the overall development process.

In Section 1.1, architecture and its meaning in software and system development will be introduced. In Section 1.2, a distinction between software architecture and system architecture is made. Essential differences are highlighted regarding needs of architecture evaluation and analysis. Architectural decisions with respect to the automotive domain are presented in Section 1.3. Section 1.4 highlights the objectives of this work. An outline is provided by Section 1.5.

1.1 Architecture in Software and System Development

Architecture is presented as accompanying activity of the development of software-intensive embedded systems in this chapter. To distinguish architecture from design and other views, especially structural ones, some definitions are given below. It is just a selection of available definitions. An extensive collection is available on the homepage of the Software Engineering Institute of the Carnegie Mellon Univer-

sity [SEI07]. Each of the following definitions will be discussed in short highlighting the most important parts with respect to software-intensive embedded systems.

> [...] the architectural design of a system can be described from (at least) three perspectives – functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure. [...]
>
> *Len Bass et al.*

This definition is already related to the architecture of systems in contrast to pure software. The structure refers to the hardware of the systems constituting the system resources. These are used by the system's functions usually implemented as software and mapped to hardware devices (the allocation). The embedded system architecture presented in Chapter 2 is based on this principle of mapping components of one view onto another one. The above definition is more general whereas the embedded system architecture is dedicated to controller network systems.

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
>
> *Len Bass, Paul Clements, and Rick Kazman*

Besides the structure, emphasized in this definition (cf. Bass et al. [BCK98]), the properties and relations among the components are mentioned. This aims to a component-oriented view taking component details (element properties) and relations into account that may motivate specific architectural decisions. Tradeoffs are often based on such relations which impede to meet quality requirements in the required or desired way and thus are the center of further investigations. Although sensitivity analysis may need additional details and models, the relations mentioned above are accountable for dependencies between quality attributes.

> A critical aspect of the design for any large software system is its gross structure that is, its high-level organization of computational elements and interactions between those elements. Broadly speaking, we refer to this as the software architectural level of design.
>
> *David Garlan*

The high-level character of architecture as part of the design process is pointed out by David Garlan (cf. Garlan and Perry [GP94]). Computation and interaction are first-class citizens in the design domain this quotation refers to. This view on a system is related to the function architecture and its demands of resources in the embedded system architecture in Chapter 2.

> Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be canceled.
>
> *Eoin Woods*

The decisions regarding the components to be combined into an architecture are in the center of interest here although components are not mentioned directly. The importance of such decisions especially as a basis for the project is highlighted. Architecture as structure as well as set of decisions attends the whole development process. The dependencies between components and decisions are mentioned in the following sections.

> Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements such as scalability and availability. Software architecture deals with abstraction, with decomposition and composition, with style and esthetics.
>
> *Philippe Kruchten*

Philippe Kruchten's definition (cf. Kruchten and Thompson [KT94]) is directed to the satisfaction of functional and non-functional requirements based on the software structure. Moreover, the abstraction by architecture and the component-based view (see Section 2.2) are brought into relation.

> Software architecture is an important level of description for software systems. At this level of abstraction key design issues include gross-level decompositional components, protocols of interaction between those components, global system properties (such as throughput and latency), and life-cycle issues (such as maintainability, extent of reuse, and platform independence).
>
> *Gregory D. Abowd*

Besides architecture as a means of description, its relation to quality attributes again is pointed out in this definition (cf. Abowd et al. [AAG95]). They are referred to as global system properties and life cycle issues but can be considered as quality attributes.

The definitions given above address various points of view on architecture as development discipline. Recapitulating, the high-level structural view as well as the reasonable combination of architectural elements are the essential parts of an architecture. One question still remains regarding architecture in the development process. This question is concerned with the differentiation between architecture and design. After all, there are also structural views in conventional design, e.g.

class diagrams. According to Paul Clements in [CBB+02], it is a matter of abstraction. Actually, design of every level can be architecture. It is the viewpoint of considering certain aspects of design which defines what architecture actually is. The more detailed a design task is, the more detailed architecture decisions can be. Architecture in a low-level design task can be much too detailed to be considered as high-level structural view of other tasks. Thus, it depends on the software and system to be developed and more precisely on the part of the development process which is considered. Instead of developing another architecture description language (cf. Debruyne et al. [DSLT04], Feiler et al. [FGH06], Dashofy et al. [DdHT01], and the Object Management Group [OMG06]), a metamodel—specific for the automotive domain—is defined in Chapter 2. It describes structural views on embedded system architecture in terms of controller networks used in the automotive domain. Because of the amount of various subdomains of the software-intensive embedded systems domain, automotive embedded systems are taken as a representative in the following, although there may be certain differences between the subdomains. A general differentiation between software and system architecture is given in Section 1.2 which, in first place, addresses the strong relation of embedded systems to their hardware.

Although architecture has a superordinate position in the development process, it will accompany the entire process partially for definition and partially for quality assurance. Thus, the view on architecture itself may change over time in a development process. Nevertheless, architecture stays the superior structure of software or a system in combination with the rationale behind. Automotive embedded system architecture will be in the center of interest in the following chapters. Several details extending the given coarse architecture definition of structure and rationale will be discussed in the context of the particular chapter or section.

1.2 Software versus System Architecture

Why to make a difference between software and system architecture? Especially with the latter one as software-intensive embedded system architecture which will be in the center of interest in the following. Both have their functionality defined by software. After all, both are executed on hardware. There are two viewpoints justifying this distinction. The first is the technological one taking the hardware into account. The second one is directed to the evaluation based on quality attributes.

Extra-functional requirements such as performance and reliability strongly depend on the underlying hardware platform. Software architecture usually does not include the hardware platform as part of the architecture. Thus, extra-functional requirements depending on hardware are harder to express and quantify in software than in system architecture. Furthermore, the hardware resources of embedded systems are way shorter than in other domains, which is because of the additional technical and physical restrictions besides the economical ones. The weight, size, resistance against

outside influences, power consumption, network connection, and so on are considerably more critical than they are in conventional systems. In addition, memory and other parts of e.g. controllers are more cost-intensive than in non-embedded use. Hence, the hardware as part of system architecture is more restricted and restrictive than the hardware as realization platform but not as part of software architecture.

An architecture evaluation reviews and assures the sufficiency of one or more architecture variants to meet the quality requirements. Thus, concrete quality attributes have to be taken into account. Against the background of software and system architecture, the orientation of quality attributes may change. For example, the portability of functionality (software) to another platform (hardware) is a quality attribute of a software user or an administrator whereas in embedded systems, it is a vendor's ones. The difference is again based on the hardware as part of the system. The embedded system is provided as a whole, i.e. with hardware like controllers, sensors, and actuators and often with network and power supply. Hence, it is not likely that an embedded system user will ever change the system except changing components at certain interfaces. The porting of software to another hardware component will be performed exclusively by the vendor. Portability is needed to support component reuse and maybe product line approaches by the vendor and therefore has a different meaning than with respect to software architectures.

It is quite important to have the information on the kind of architecture to be evaluated for the given reasons of technology and quality attribute orientation. Evaluation as a communication vehicle being part of architecture documentation (see Clements et al. [CBB+02]) has to be understood in the appropriate context. The interpretation of results can be quite different and even more the consequences on further development of an architecture need a proper understanding of the background of past architectural decisions and their evaluation results.

1.3 Architectural Decisions

Building an architecture is, like the whole software or systems development process, based on various decisions. Thus, architecture as structure or structures of a system (cf. Clements et al. [CBB+02]) is based on structural design decisions regarding the components and the way in which to combine them into a system or subsystem. These decisions are called architectural decisions. Although the structure should be determined before building the system, this is not done in a single step. Architectural decisions depend on earlier decisions and may influence later ones. In this section, the levels of architecture decisions according to the point in time to be made and their concerns are discussed. A first rough distinction can be drawn between top-level, high-level, and low-level architectural decisions. Figure 1.1 shows a sequential order from top-level architectural decisions down to low-level ones. Actually, neither the sublevels nor the levels themselves are totally independent. The sequential order

shows the chronology of the main levels' decisions. Sublevel decisions often may be made in parallel or overlapping in time. On the right hand side of Figure 1.1, a distinction between the impact of the levels' decisions (see Section 1.3.1) and the impact predictability (see Section 1.3.2) is depicted. The distinction of decisions shows not only the differences of levels but presents which ones are considered as architectural decisions and therefore partially define the architecture of a system.

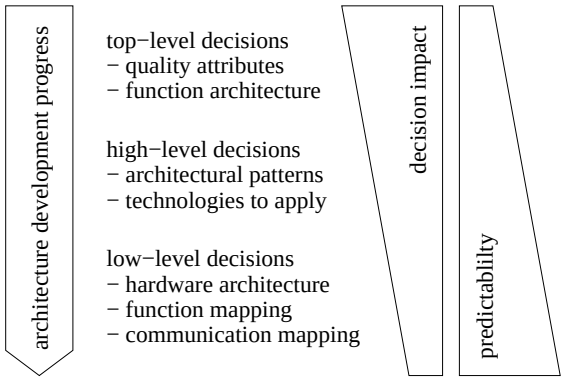

Figure 1.1: Decision levels

Top-level architectural decisions concern the functionality of the system to be realized. Functional requirements are part of the functionality. Extra-functional requirements are represented by quality attributes. The choice of quality attributes as well as their importance are assigned to the top-level, too. Thus, what the software or system is meant to do and how well it is meant to do it are top-level decisions.

Especially in evaluation processes in which several architecture variants are taken into account, a common denominator is needed for a fair comparison. The functionality stated by top-level decisions builds this common denominator. As one of the earliest choices, it is invariant and has to be respected by all subsequent decisions except for the choice of quality attributes which can be seen as orthogonal. Nevertheless, the quality attributes as extra-functional requirements should be suitable regarding the functional requirements of the chosen functionality.

High-level architectural decisions are those considering the application of architectural patterns (see Buschmann et al. [BMR+96, BHS07], Schmidt et al. [SSRB00], and Kircher et al. [KJ04]) and various technical options in order to meet architectural requirements represented by main quality attributes. Thus, high-level decisions are mainly trend-setting ones. They are to be made early in the architecture development process, therefore influencing most of the subsequent low-level decisions.

Different high-level decisions are those creating most diversity in architecture variants. A commonly known type of decision is the number of controllers to be applied in the architecture. This can lead to a more centralized or more decentralized architecture, which restricts or at least influences nearly every subsequent decision. Other common representatives are the communication technology (e.g. digital against analog, various bus systems), the power supply (installation of a 42 volts electrical system), and the processor architecture (8, 16, and 32 bit).

Low-level architectural decisions are closest to the system's realization. They deal with the choice of the system composition which has to be done following the top and high-level decisions. In Chapter 2, views on the architecture of embedded systems are presented which mirror the sublevels of low-level decisions. The first sublevel is the choice of hardware components, i.e. controllers, sensors, actuators, and communication lines and how to connect them. After the so called hardware architecture has been figured out, the functionality has to be distributed to the hardware, which is the second sublevel. This function mapping determines which functions have to communicate across controller boundaries. The third sublevel is the communication mapping taking the inter-controller communication needs into account.

Actually, the architecture may be defined at this point but the system still needs further development which will be based on all previous decisions. Hence, further steps are referred to as design instead of architecture. Even though many decisions usually made in design are stated already in architecture or as architectural decision in this chapter, implementation and configuration tasks still have to be done and still have the chance to improve or spoil the system. In this section, a differentiation between architecture and design has been addressed by explicitly denoting and dividing architectural decisions in certain levels. Further decisions, usually much more detailed and accounting platform specific properties, can be considered as design decisions without architectural background although earlier architectural decisions have to be taken into account, of course.

1.3.1 Decision Impact

The *decision impact* is the quality expectation of a defining decision in building an architecture. For later development and reuse of architecture, the difference of quality caused by a change of an architecture can be understood as impact, too.

The decision impact is greatest for earliest decisions (cf. Bontempi and Kruijtzer [BK04]), i.e. top-level directly followed by high-level decisions. Subsequent ones will depend on these and are restricted because they have to take earlier decisions into account. As a consequence, incorrect early decisions bear many problems and are the hardest to fix. Top-level decisions which lead to concrete functional and extra-functional requirements are the most far-reaching. Actually, building a system

without making these decisions, i.e. without requirements, is trivial as every realization of a system can meet no requirements (cf. Clements et al. [CBB+02]). At this point, the legitimate question arises, how top-level decisions can have impact and on what. After all, they set the requirements and thus have been made from scratch. There are no concrete requirements to be taken into account by top-level decisions. But, the functionality to be provided and the demanded quality represented by quality attributes are not given as arbitrary choices by some management. The superior goal is to build systems which are realizable, attractive, and competitive on the market. Thus, top-level decisions are made with a superior goal in mind.

High-level decisions restrict the options for later ones. They fill the gap between the abstract top-level and the concrete low-level. Consequently, the impact of such decisions is quite high, hence incorrect decisions may be fatal for the whole project. High-level decisions are the first to induce a number of architecture variants. Usually, these variants follow one or more strategies to meet certain requirements. Tradeoffs between different quality attributes are often regarded at this level. Thus, high-level decisions often prohibit a development in directions other than the intended. Therefore, most variants are disjunctive regarding their strategies they embark on. The decision impact on this level can be compared to the success of the strategies in meeting the quality goals.

Low-level decisions still have enough impact to support high quality systems as well as to spoil the system. They can be likewise fatal but are easier to revise because the main directions of the development are already stated by higher levels' decisions and rethinking low-level ones does neither include withdrawal of earlier decisions nor accounting new strategies. Thus, the choices are more restricted than on higher levels. Nevertheless, even low-level decisions are still architectural and state the structure of embedded systems in this case. Thus, changes can still be cost-intensive, especially if parts of the architecture are already realized and for example fully configured controllers have to be changed to add unattended or new functionality.

1.3.2 Predictability

Knowing about a decision's impact is necessary to avoid surprises in architecture development. To really benefit from this knowledge, the impact needs to be predicted. In which direction and even more how much will a decision impact the architecture quality? The higher the predictability, i.e. the more precise, the greater the benefit for the development process. But, it is not just the ability to predict that is interesting. The predictions' reliability has to be taken into account and therefore is a part if the predictability, as well.

The predictability regarding different decision levels is contrary to the impact itself. The earlier a decision is made, the more impact it has, the less predictable it is. This is due to the fact, that there is a wide range of consequences a decision can have based on the many options not yet chosen by subsequent decisions.

The predictability of top-level decision impact is based on experiences concerning the buildability of the systems and tradeoffs of quality attributes involved. Hence, predictability statements on this level are limited to an estimation of the harmony of requirements stated by the level's decisions. Unrealistic requirement compositions can be avoided. Actually, the prediction of the decisions' impact is interlaced in the decision-making process, not appended to the process. The top-level's process is more iterative than other levels' processes because it bears not a number of variants but a single set of requirements on which no selection is performed. Especially as most fundamental, requirements need to be reasonable, there is no alternative to choose in the end.

High-level decisions again fill the gap between top and low-level decisions. Their impact can be predicted easier because the results of applying certain architectural patterns or technologies are well known. Actually, the patterns and technologies are defined and developed to support selected requirements. Predictability on this level covers the impact direction, i.e. if certain qualities are likely to be fulfilled. Concrete statements are rare on this level.

The impact of low-level decisions is the first to be predictable more precisely, the reason for which is the strong relation to the system realization. It can be put directly into relation to one or more quality attributes (mostly subattributes of the main attributes). Thus, the impact of a low-level decision as a determination or change of the architecture can be expressed by means of sensitivity analysis. Therefore, dependencies of quality attributes on the architecture can represent the impact of low-level decisions. These concrete dependencies allow quite precise predictions.

1.4 Objectives

The main objective of the approach presented in this work is to provide decision support for embedded system development in the automotive domain. By modeling the architecture of embedded systems, the essential properties of the systems are predefined and can be evaluated. Architecture variants represent the decisions made on how to build a system. Evaluation—with respect to explicitly modeled quality requirements—results not only in statements on the quality of the architecture variants but rather in statements on the decisions represented by the variants. Thus, the impact of a decision can be identified. The feedback of decision impact can be used to improve the architecture quality as well as the development process in terms of supporting future decisions based on the experiences made. The predictability of decision impact will be raised, which increases the reliability of architectural decisions.

The more accurate and reliable the impact of decisions and the quality of an architecture can be evaluated, the less development effort will be spent on insufficient architecture variants. Furthermore, avoiding suboptimal decisions saves valuable

development time, which is quite short especially for embedded systems as part of an overall system to be developed.

As a great share of the functionality of an embedded system in the automotive domain will be required in future products as well, reusability not only of parts of the system but rather of architectural decisions made during the development is highly desirable. Development experiences can be expressed in terms of good and bad decisions made with respect to specific requirements. Therefore, an investigation of architectural decisions by analyzing the results of architecture evaluation will be performed.

The quantification of evaluation results based on an explicit consideration of hardware as part of system architecture can be used to make evaluation results comparable with respect to different architecture variants and even specific decisions. Thus, tradeoffs and relevant dependencies in the architecture as well as between architecture and evaluation results can be identified. Furthermore, quantification supports the analysis for architecture potential in terms of design space and even architecture space exploration. Thus, decisions of how to change an architecture can not only be supported but rather be documented, which is an essential part in making decisions easy to describe, justify, and reuse.

All in all, knowledge of architecture quality as well of the concise documentation of architecture requirements and adequate architectural decisions will help to improve the development of embedded systems. Effort spend on insufficient alternatives can be saved and successful developments can be reused to save costs and time which both are quite short in development processes.

1.5 Outline

Chapter 2 contains an introduction on architecture modeling in terms of a metamodel for embedded systems in the automotive domain. This metamodel defines views on the architecture as basis for modeling and evaluation of architecture variants. Two case studies of the automotive domain are modeled in Chapter 3. The main decisions represented by the variants are discussed in detail. A metamodel for evaluation structures, called Quality Attribute Directed Acyclic Graph (QADAG for short), is introduced in Chapter 4. An instance of the QADAG for one of the case studies is constructed to illustrate how to use the QADAG to model architecture requirements. Important dimensions of architecture evaluation are introduced in Chapter 5 as basis for defining evaluation tactics as part of a methodology for time-saving and cost-efficient evaluation processing. An example for application of the tactics is given with respect to the evaluation of one of the case studies. The case studies presented in Chapter 3 are evaluated in Chapter 6. Parts of the input needed for evaluation is given in Appendix A. The results of the evaluations are contained in Chapter 6 as well as in Appendix B. Dependencies of the architecture

evaluation and resulting sensitivities of the evaluation results on architectural decisions (i.e. changes) are introduced in Chapter 7. Architecture analysis based on the architecture requirements—represented by QADAG instances—and on the evaluation results is presented in that chapter as well. In Chapter 8, the case studies are analyzed by the analysis approaches of the previous chapter. The analysis results are discussed with respect to decision support in terms of how to proceed in the architecture evaluation. Chapter 9 concludes.

2 Embedded System Architecture Views

Views on embedded system architecture at controller network level will be presented in this chapter. These views have been motivated by the automotive domain. They are not meant to represent the system in every detail but to represent architectural variants and thus decisions, i.e. in which way the structure has been determined and which components are applied. Actually, most details are not available right from the beginning. Thus, the modeling of the views presented in this chapter is extendable to be capable of representing details available at different points in time in the development process.

As already stated in Section 1.1, controller networks used in the automotive domain are taken as representative for software-intensive embedded systems. The development process is usually oriented towards the controllers in this domain but more flexibility in distributing the functionality as well as the reuse of hardware and software is aspired. The benefit of a change of the development process to a more function-oriented one are the decrease of the number of controllers needed to build the system in combination with less development time mainly based on reuse (cf. AUTOSAR, see Heinecke et al. [HSF+04]). An explicit decoupling of hardware and software is necessary to achieve these goals. The views defined in this chapter are meant to provide this distinction and to express differences in architecture variants. Hence, the development process can be oriented towards the functions of the system to be built instead of towards given controllers providing the functions.

In Section 2.1, fundamentals of architecture modeling are presented. The underlying component-and-connector viewtype (C&C, see Clements et al. [CBB+02]) is introduced in Section 2.2. All views—as parts of the embedded system architecture—are based on this viewtype and thus represent the connection of components on a certain level. Section 2.3 presents the function architecture which is a C&C view on the functionality provided by a system. This functionality is assumed to be invariant in different architecture variants thus stating a common basis for comparison. The hardware architecture in Section 2.4 provides a view on controllers, sensors, and actuators connected via communication lines building the controller network. The mapping of functions to controllers is content of Section 2.5. The ability of separately developing hardware and software is one of the main advancements explicitly taking the architecture of a system into account. After functions have been mapped, the inter-controller communication can be derived from the function mapping and be mapped to communication lines, which is described in Section 2.6. Section 2.7 addresses the visualization of architecture. Architecture examples are given in the automotive case studies in Chapter 3.

2.1 Architecture Modeling

Some of the terms of architecture used in the following sections are given below although they are introduced in detail in their respective context. Most of the terms may be generally known but need to be brought into relation with architecture modeling to avoid misunderstandings.

2.1.1 Types, Instances, and Refinement

A *type* predefines the content, the signature, and sometimes the behavior of architecture modeling elements. Thus, a type may be considered as more or less complex data structure for modeling purpose. The content is meant to denote the attributes of an element whereas the signature denotes the interface, i.e. how to access the element in an architecture. If the architecture element is meant to have behavior, this may also be predefined by the type. The behavior describes the element's activity in runtime and its reaction to stimuli. All in all, a type compounds similarities of its elements. The *hierarchy of types* describes their inheritance, i.e. generalization and specialization. The direction of specialization focuses the *refinement* of a type by adding further detail to the original type definition. The type of the type of some element (or even type) is called its *supertype*.

The *instance* of a type is a concrete element implementing the definitions of its type. It has *properties* according to the attributes defined by the type. If the element has runtime presence, its behavior may be specified if not already done by the type. In this context, *runtime presence* is meant to be the existence of the element as such in runtime, i.e. there is a counterpart in the runtime system for the modeling element of the design time system.

2.1.2 Viewtypes, Styles, and Views

To emphasize particular points of interest of a complex system and its structure, different *views* on the system may be useful. In order to highlight the points of interest, a view needs to abstract from details not in the center of interest. Clements et al. state three types of views useful in software architecture in [CBB+02] called *viewtypes*. These viewtypes are directed to implementation units, to elements with runtime behavior and interactions, i.e. runtime presence, and to non software structures. They are called the module viewtype, the component-and-connector (C&C) viewtype, and the allocation viewtype.

The viewtypes predefine only the focus of the view, e.g. the decomposition of a system. To actually define the view in detail, a so-called style will be given (in the notation of UML class diagrams, see [OMG03] and Jeckle et al. [JRH+04] in this work). A *style* can be considered as application of the viewtype for special purpose, i.e. to highlight architecture concerns by defining a view based on the viewtype.

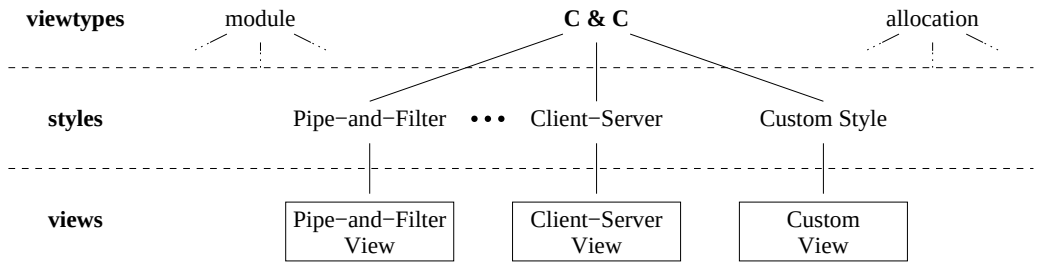

Figure 2.1: Viewtypes, styles, and views, cf. [CBB+02]

Thus, a style is a concrete definition of a view in terms of a given viewtype. The refinement from viewtype via style to a view is depicted in Figure 2.1 (cf. [CBB+02]). An example for a style based on a viewtype with decomposition purpose may be a systems decomposition of the hardware. Another one may be the decomposition of the software. Thus, separate views on the system hardware and software can be provided by views which are predefined by the styles.

The Custom Style represents the styles defined in Sections 2.3, 2.4, 2.5, and 2.6 especially for the purpose of providing views on automotive embedded system architecture and their variants. In terms of types as discussed above, the styles provide the refinement from viewtype to view. As will be discussed in Section 2.2, the C&C viewtype is chosen to be the viewtype as basis for defining the views on embedded systems in this approach. The terms component and connector will be explained below in order to increase the understandability of the following sections.

2.1.3 Components and Connectors

For the description of the structure of a system, a decomposition is necessary. The elements of which the system structure will be composed are components and connectors according to the C&C viewtype. The *components* represent the part of which a system is composed whereas the *connectors* describe how the components are put together, i.e. are connected one to another. Components as parts can be held in a library on their own whereas connectors need the context of the components to be connected. Hence, they cannot be held in a library on their own. Dependent on the component types to be connected, the connectors have a special concern. They are not meant to be just links between components but to describe connections in detail according to the style currently applied.

2.1.4 System and Environment

With respect to architecture modeling in this approach, a system consists of the software realizing the functionality of the system and the hardware on which the software will be implemented. Furthermore, the sensors and actuators as well as communica-

tion lines are components of the system. Views on a system often contain the system interface to its environment. The interface to the environment of a system needs to be taken into account with respect of the meaning of environment. First, a system as subsystem of a bigger one has an environment of similar systems. The interface to this part of its environment is built by the elements of the system and their interfaces. In embedded systems in the automotive domain, such interfaces usually are realized via buses. Second, a system as embedded part in an environment needs to provide additional interfaces to that non-similar part of its environment, i.e. the non-electric and non-electronic parts of the environment like mechanical parts and devices. Those need e.g. an actuator driving them. Thus, the sensors and actuators of the system build the interface to the environment of the embedded system. In the automotive domain, power window systems contain a simple example for such interfaces. The actuator to drive the window is a motor and actually belongs to the embedded system. The window itself contains no interface to the system. Nevertheless, the power window system needs to know when the window is e.g. closed. Sensors—again part of the system—build the interface of the system to the window checking its status. The connection between the sensors and the window are not considered as part of the embedded system. The views on embedded system architecture defined later will contain all elements necessary for modeling the interfaces to the environment.

2.1.5 Functionality, Functions, and Features

The *functionality* of a system describes what the system is meant to do. In software-intensive embedded systems, the functionality of the system usually is realized by software. There are some exceptions in which small parts of the functionality may be realized in hardware but these are considered as auxiliary functions and thus are not taken into account any further. Like a system can be composed of subsystems, its functionality can be composed of the subsystems' functionalities. Nevertheless, different views on the functionality can be provided regarding its realization by software components and its noticeability by the system user, which in fact are not uniform. In this section, both viewpoints are discussed with respect to their usage in this approach.

Functions in terms of software components are needed to decompose the functionality in order to distribute it among hardware components. Thus, for the use in the views on embedded system architecture, which will be defined in Section 2.3, functions are software components to be distributed. An adequate level of abstraction needs to be found to achieve an applicable decomposition and distribution. Hence, the lowest level will be software components or sets of them that can be run on a single controller, i.e. the smallest unit to be mapped. This lower bound assures that only executable distributions can be modeled. The upper bound of functions should be set according to provide enough composition to reasonably distribute the functionality among the controllers. A composition of software components to a single function

can be useful as long as their combined mapping is intented by the function designer. Especially in cases in which the system functionality is extended, the question arises whether to extend an existing function or to define an additional one. An example for this scenario is the extension of a power window feature by a Short Lift Control function. This function extends the original one by realizing short movements of the window to release it from its groove, which is necessary when the door is opened and the window is not framed by the door. In the case study in Section 3.1, the Short Lift Control will be modeled as an additional function because it may be mapped to another controller than the original Power Window Control function. In future projects, a common mapping may be intented. Short Lift Control may become part of the Power Window Control function in other projects.

The explicit modeling of tasks, processes, and software components like drivers etc. as parts of functions is omitted with respect to functions as distributable parts of the functionality. Although such a refinement may become interesting for further analysis of the system and later implementation, it is not raised to the architecture level of this approach.

Features as part of the functionality of a system are meant to be user noticeable. The user of the system will not be aware neither of most parts of the hardware nor the software and especially not of their decomposition. Thus, a feature is no direct composition of functions although it is realized by them. Moreover, the users of the system are not standardized. What a user may notice and what seems to stay unnoticed strongly depends on the actual user. Therefore, the identification of features may be less easy than the definition of functions. An example for the derivation of features from functionality may be the feature list from which to select when ordering a new automobile. Power windows build a feature as user noticeable part of the functionality. The Short Lift Control does not while it usually is not noticed as part of functionality on its own.

Although the terms functionality, feature, and function are settled now, the meaning of these terms becomes even more complicated trying to set them into relation. At least when variant, version, and configuration management comes into view, one and the same feature of different systems is not necessarily built of the same functions. Moreover, the same function may be used to realize different features.

The view on functionality decoupled from the hardware allows for changes in the development process in order to provide more flexibility in distributing the functionality as well as the reuse of the hardware and software components as aspired by e.g. the AUTOSAR development partnership, see Heinecke et al. [HSF+04]. While current development is oriented towards controllers, which provide parts of the functionality mostly directed to features, the decoupling of hardware and software provides the distribution of functions to selected controllers. Besides increased flexibility and component reuse, fewer controllers may be needed to realize a system and the development time may be reduced as well. Hence, function-oriented development

processes are the upcoming way which need to be supported by appropriate views on the system architecture as will be defined in this chapter.

2.1.6 Architectural Decisions and Variants

One of the main purposes of the approach of the current work in modeling architecture is the representation of architectural decisions on the basis of architecture variants. An *architecture variant* is built to provide the requested functionality and meet the quality requirements stated by quality attributes. An *architectural decision* is meant to be a decision of how to build or rather change an architecture. If several variants of architectures are available, the choice of one of them may contain several decisions. A decision does not necessarily concern the whole architecture but can be focused on certain areas. Hence, variants with respect to specific decisions regarding only certain areas are taken into account as well. Different levels of architectural decisions are presented in detail in Section 1.3.

2.2 Component-and-Connector Viewtype

Architecture is meant to be the structure or structures of software or a system (see Section 1.1). These structures are based on a decomposition of the system into components describing the parts the system is built of. Furthermore, connectors are needed to describe how the components of a system are composed to a system or maybe part of it. The component-and-connector viewtype, C&C viewtype, is not just meant to describe components of a system and how to put them together by using connectors but even emphasizes the runtime presence of its elements (see Clements et al. [CBB+02]), which is conform to most of the common views on embedded system architecture. First, the elements have runtime presence because they are elements of models of the systems structure. This structure usually does not change during runtime, i.e. the structure is static, not dynamic. Thus, the elements defined in the architecture design time have a counterpart in runtime. Second, the properties of those elements usually are related to resource consumption (memory, bus capacity, power, etc.). Those resources are consumed at runtime. The concern in building embedded system architectures is focused on resource allocation. Although there are several other concerns, most of them are at least indirectly related to resource consumption. One example is the extendability of a system. Not just interfaces to new functionality but also resources reserved and available for performing that additional functionality build the center of interest. Another example is reliability. As important input for reliability evaluation, the mean time between failures (MTBF) of components, is related to runtime.

Figure 2.2 shows the C&C metamodel used to define different views of the viewtype. In Sections 2.3 to 2.6, the views needed for embedded system architecture in

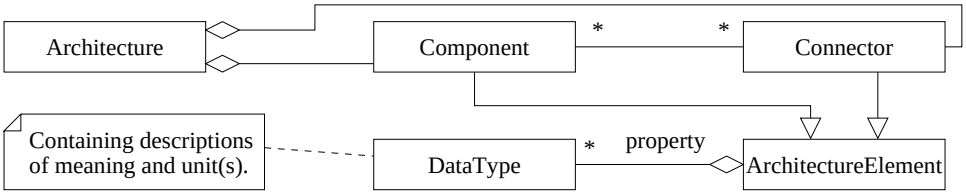

Figure 2.2: The component-and-connector viewtype

the automotive domain, first defined in Florentz and Huhn [FH06], are presented as styles. They are kept abstract to provide most universality and allow adaption to the needs of architecture development in different states. Extensions of the metamodel—actually the styles—can be used to refine the views. As automotive embedded systems are taken as example, the views represent embedded systems based on controller networks.

It is a frequently asked question which element to be a component and which one to be a connector. The answers will be given in the respective sections in combination with a detailed description of the intention of the views and the elements shared by them. According to Clements et al. [CBB+02], a table containing summarized information on the view is given at the beginning of the description of the respective view. Nevertheless, some general arguments regarding components and connectors are brought forward. First of all, components stay components in all views. There is neither a change of an element from component to connector nor in the other direction across views. This is important to keep clarity of the views as well as of their instances, i.e. architecture models. Second, components are meant to be available in a digital library and thus are invariant regarding their application in an architecture variant. Connectors, which define how an architecture is put together, will not be available in libraries because they are not expressive without their component context. To build that context, connectors are essential in every architecture view. David Garlan states the great importance of connectors in [CBB+02] especially in terms of their complexity, i.e. connectors are not meant to be only trivial connecting elements but rather essential elements of C&C views.

Figure 2.3 outlines the four views on the embedded system architecture presented in the following sections. The connectors of each view are represented by hexagons.

1. The first view is concerned with the connection between functions, more specifically between required and provided signals of functions.

2. The second view takes hardware connectors between hardware devices and communication lines into account.

3. The third view represents the function mapping.

4. The fourth view is the communication mapping.

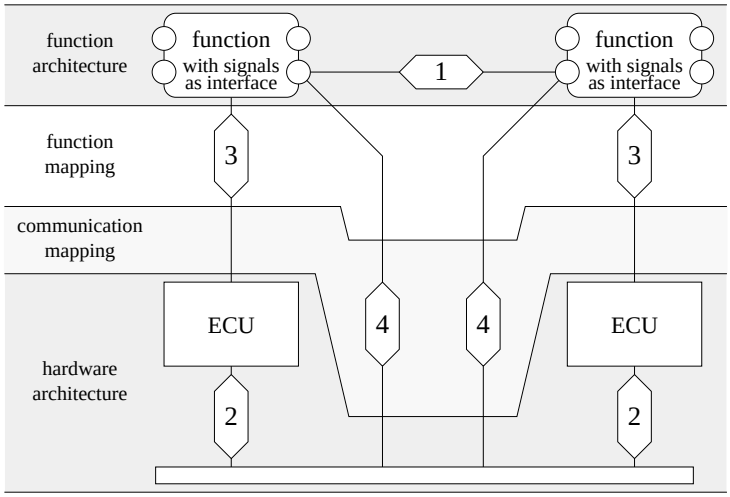

Figure 2.3: Scheme of views defined for the embedded system architecture

The function architecture and the hardware architecture describe which components are to be put together and how this should be done. The function mapping and the communication mapping are C&C views as well. But, they do not make a choice which components to apply but how to connect components already deployed in other architectures (i.e. function and hardware). Hence, the mappings are mainly based on the connection of the overall embedded system architecture. This central use of connectors in the mapping again highlights the importance of connectors as essential elements in architecture modeling.

The elements of Figure 2.3 will be mapped to the Open Systems Interconnection (OSI) reference model in Figure 2.10 of Section 2.6.

2.3 The Function Architecture

The functionality of a system is described by its function architecture. Functions can be considered as software components requiring input and providing output. A function's computational behavior may be defined by code, behavioral models (e.g. state-based formalisms), and even by textual descriptions. The architecture model is meant to represent the functionality structure composed of software components, i.e. functions, which will be mapped to hardware later. Hence, the actual computational behavior is not in the center of interest in this view on architecture. After all, it becomes interesting when the functionality finally is mapped to the hardware platform. Then, first simulations can be performed and communication behavior becomes observable. For architecture purposes, the functions represent parts of the functionality and define their signature, i.e. the input and output. Both of them are refined by

required and provided signals as parts of the functions. In order to connect functions, their signals are connected. Thus, the connectors of the function architecture are directly linked to signals and only indirectly to functions. Figure 2.4 depicts the metamodel of the function architecture view. Its essential elements are summarized in Table 2.1. A more detailed description of the elements, the construction, and the application of the metamodel is given below.

components	
functions	A function represents a software component as a part of the system functionality. The computational behavior of the function is not necessarily included as executable model in the function but it has to be known at least informally. The signature of the function is given as its required and provided signals.
signals	A signal represents information to be communicated. It is specified by its name, content, availability and resolution.
connector	
signal-to-signal	A signal-to-signal connector describes how to connect two signals. Because signals are used to define a function's signature, a signal-to-signal connector indirectly connects functions by describing how a provided signal of one function is matched with a required signal of another one. Signals provided by sensors and required by actuators are taken into account as well.

Table 2.1: The function architecture

2.3.1 Functions

For evaluation purposes, the functionality of a system is invariant. Variations of functionality over the architecture variants to be evaluated may cause an inequitable context for the evaluation results. Determining the functionality, i.e. building the function architecture, is a top-level decision in architecture development (see Section 1.3). From the user's viewpoint, functionality is built of features which can be considered as being user-noticeable, e.g. power windows, adaptive cruise control, and so on. Actually, the decision of the system functionality is based on features selected by e.g. a salesman or a marketing team. Those features to be realized by a system determine the functions to be mapped to the hardware architecture and vice versa.

Thus, features provide a view on the system different from that of functions but the resulting functionality is the same. The intent of the embedded system architecture presented in this chapter is to point out architecture decisions. Thus, the view on the system functionality is directed to the functions and not to the features in order to be able to express variants of the distribution of the functionality among the controllers (see Section 2.1).

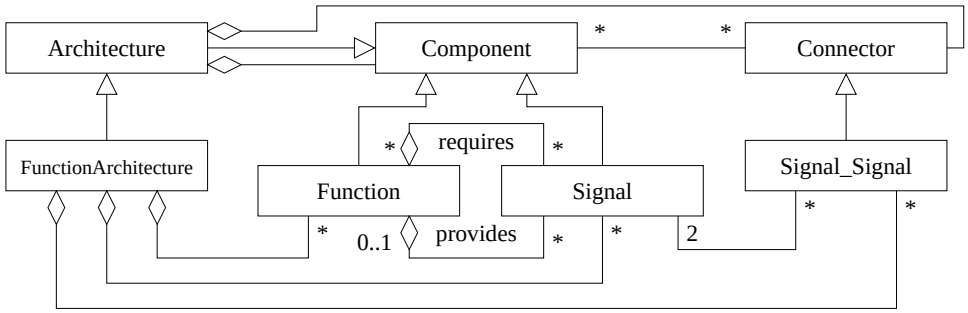

Figure 2.4: The function architecture metamodel

As a software component realizes a part of the functionality, functions have computational behavior requiring input and providing output. Although the computational behavior itself is not in the center of interest of this view, however, the resources needed for their computation are. Therefore, the requirements regarding the ROM, RAM, and CPU capacity need to be expressed as properties of a function to allow for evaluation of the mapping of these functions to controllers. Requirements regarding input and output are considered by the required and provided signals assigned to a function. Both, functions and signals, are components. Hence, their direct relation is not in the scope of C&C viewtype, i.e. they are not connected via a connector but just associated. They build structurally complex components.

2.3.2 Signals

The term *signal* is often used to describe the message transmission between a sender and a receiver. The actual realization of a signal depends on the application domain. A signal can be meant to be broadcasted to a set of receivers. For example, it can be meant to be applied voltage (or maybe its changes) on a wire connected to hardware devices. In terms of defining a function, signals are meant to be required input and provided output. There are several different and domain-specific interpretations of signals. The commonality of all the interpretations is that information is communicated by signals. This is what is expressed by a signal in the metamodel given above. For architecture description, this common interpretation covers all needs. The specific interpretation of a signal is covered by the communication mapping (see Section 2.6, especially Figure 2.10).

The specification of a signal contains its content, its availability, and its resolution in special cases. The content is considered to be the information represented by the signal. Besides a name and an informal description, the width in bits and their arrangement have to be specified, i.e. the format. The availability describes the point(s) in time in which a signal is available namely the rate of its availability. In case of a required signal, the availability expresses a requirement as well. The resolution of a signal sometimes is called granularity and addresses the represented information. For example, the position of a window can be given as opened, closed, and somewhere in between. A higher resolution may provide the windows position as a metric value of the range of the window movement in centimeter or even millimeter.

2.3.3 Signal-to-signal Connector

As described above, signals represent information to be communicated. The functions to be indirectly connected via the signal-to-signal connectors usually are coordinated, i.e. their required an provided signals are specified to match when the architecture is composed. Thus, signal-to-signal connectors are trivial connectors at most, referencing the signals to be connected. In case of uncoordinated functions as well as sensors and actuators, an interpretation of the signals' content becomes necessary to match the content of the provided signal with that of the required one. A simple example may be a door latch sensor providing a signal of a logical `1` meaning the door is opened and a logical `0` meaning the door is closed. A function which requires the signal is defined to interpret the `1` as closed and the `0` as opened. The provided and required signals both have the same content and bit width. A simple connector may lead to confusion if no logical interpretation is provided. Actually, the connector has to toggle the bit to ensure the intended communication. A more complex example may be given by a function providing information on the incline of the vehicle in forwards and sidewards direction, which is needed to adjust the curve lights according to the steering angle (another signal). Some function may just need one direction of incline but both are provided in combination. The signal-to-signal connector can now be used to express the part of the provided signal matching the required signal and vice versa. Thus, uncoordinated functions can be connected via signals. Reusing functions or even bigger parts of legacy systems often requires this option.

Signals constitute the interface of a function. With functionality as central part of function-oriented system architecture, the functionality's interface has to be provided to and required by the environment of the system. The interface towards the environment is realized by sensors and actuators (see Section 2.4) which provide and require signals. A detailed description of the system interface of a system is content of Section 2.6.3.

2.3.4 Types and Instances

In data flow-oriented domains like embedded systems, a type and instance related view on modeling elements often is disregarded. First, the instantiation is done implicitly while building the model which actually is static. Thus, no changes during runtime have to be taken into account at design time. Second, libraries of predefined elements are often just sets of blocks providing specific behavior or functionality. The required input and provided output is expressed by a name and the number of bits for each of the blocks. Besides hierarchy, there is no complex composition like signal-based interfaces for functions. Those types of blocks usually are dragged to the current model and are connected output to input. A fully functional system can be build this way without even missing a types and instances view on the system components.

Although the systems to be built in the automotive domain are still static, the reuse of complex components especially with high interactivity can benefit from thoughtful type definition and instantiation mechanisms supported by a digital library. The advantages are listed below.

- The modeling of complex components lead to high modeling effort. Reuse of these components can save much effort.
- Reuse of components leads to lower diversity of components in a system, and following, less complexity of the system itself. Understandability of the systems structure and behavior is increased.
- A centralized and global definition of functions and signals allows more powerful conformance checks on interfaces, i.e. the signals. Local definitions carry no information on this conformance and thus do not support automatic checks.
- In contrast to local definitions, global ones build an anchor point for variant, version, and configuration management. Those kinds of management become more and more important for model-based system development.

A power window system needs an instance of the same control function for each of the windows to be powered. For brevity, all power windows are meant to work alike. Besides a coordination function to handle e.g. priority of different bush-buttons and switch-keys, no further differences have to be taken into account. Thus, a function type Power Window Control needs to be defined. It requires and provides signals that have to be defined as well. Because of the Power Window Control being still abstract, neither concrete input is required nor output provided. It depends on the application of the instances which of the signals need to be connected. But, the signal types can be defined already, e.g. opening, closing, and locking of a window. Hence, for function instances, signal types are known which is needed by and sufficient for compatibility checking. To get to the point, there should be no separate power window function

in the library for each of the windows if the functions are alike. The only reason for that may be to assign more concrete, i.e. initialized signals as interface, which usually is done during instantiation. The instantiation should contain (re-)naming of the signals for a better identification. However, this is not necessary for identification by the systems itself, for which signal-to-signal connectors are used. Nevertheless, a serious naming is the basis for documentation of the architecture.

The types and instances discussion provides another argument for the choice of component or connector as supertypes of the architecture elements. Components are the elements which can be predefined for later instantiation. Connectors are the elements which cannot be predefined because their existence outside a concrete context does not make any sense. This context is provided by the connected components.

2.4 The Hardware Architecture

The hardware platform of a system is described by the hardware architecture. It consists of devices, namely controllers, sensors, and actuators, as well as of communication lines. Controllers are devices that can carry functionality, i.e. functions are mapped to them. Besides available (restricted) resources of the hardware, the computational behavior is mostly given by the functions. Sensors are meant to provide input (actually their output) for the system from the environment whereas actuators are meant to require output (actually their input) of the system to its environment. Thus, sensors and actuators build the system interface to the environment. Ignoring signals for the moment, the components of the hardware architecture can be divided into devices and communication lines. The latter are meant to transmit information to be communicated between devices. Therefore, the connectors of the hardware architecture, the device-to-commline connector, connect a device with a communication line and never two elements of the same component subtype. This part of the architecture deals with the hardware platform on which the functionality of the system will be realized. The signals—as shared components with the function architecture—build the interface to the functionality. They represent information provided and required by the system environment. Figure 2.5 depicts the metamodel of the hardware architecture. The device types of the hardware architecture are presented in Figure 2.6. Its essential elements are summarized in Table 2.2. A more detailed description of the elements, the construction, and the application of the metamodel is given below.

components	
controllers	A controller is a device to which functions can be mapped to. Because of the diversity of controllers used in embedded systems even in the same domain, no refinement is defined by this metamodel. The available resources like RAM, ROM, and CPU capacity can be given as properties. If modeling of highly complex controller structures becomes necessary, refinements can be done accordingly.
sensors	A sensor is a device which provides signals. Hence, it is a source. It represents an interface to the embedded system environment.
actuators	An actuator is a device which requires signals. Hence, it is a drain. Like a sensor, it represents an interface to the embedded system environment.
signals	Same as in function architecture (see page 21).
communication lines	As a piece of hardware, a communication line is a component of the hardware architecture, not a connector. It may be a simple communication line with just one source or a bus to which several sources can be connected. Its properties, e.g. diameter or transmission rate, depend on the actual technology applied. A communication line may even supply power.
connector	
device-to-commline	A device-to-commline (short for communication line) connector connects devices with communication lines. For example, the connector may contain information on device-related priority in case of a connection to a bus.

Table 2.2: The hardware architecture

2.4.1 Controllers

With respect to the runtime behavior, controllers provide one part of the most interesting resources for performance analysis. Controllers host functions which require these resources for being performed on an adequate level of quality. The knowledge about resource restrictions is one advantage of embedded system analysis, although

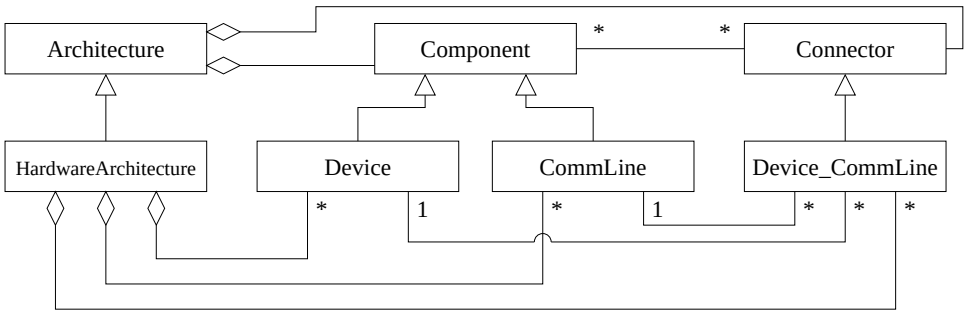

Figure 2.5: The hardware architecture metamodel

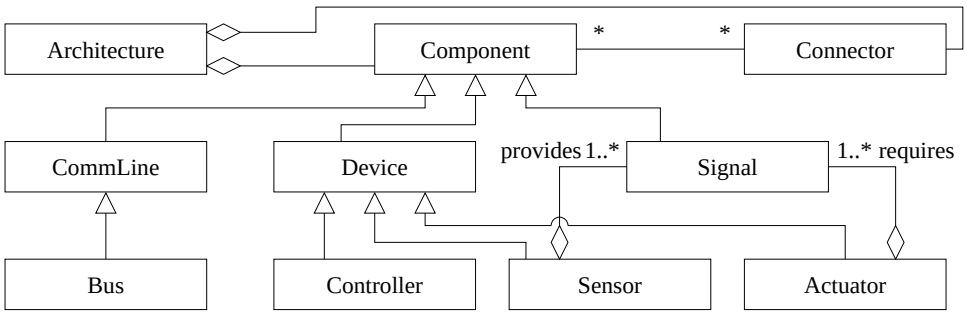

Figure 2.6: The components of the hardware architecture

the restrictions usually bear the challenge in the development of embedded systems. A controller's ROM and RAM (in most cases) are allocated statically. Thus, no specific runtime behavior is expected regarding those resources. Dynamically allocated resources like the processor are very important for performance analysis. Besides the communication configuration of the network (see below), the processor utilization significantly influences the actual performance of the system.

For the hardware architecture, the refinement of a controller by adding its properties regarding resource capacities is sufficient. For the architecture of a controller itself, explicit modeling of its components may become interesting. However, the level of evaluation and analysis applied in the following does not require such a detailed modeling of controller-intern components. The mapping of functions, which is addressed in Section 2.5, and the resulting resource usage under considerations of environment and user behavior (say scenarios) build the center of interest for performance evaluation (see Chapter 4).

Because of the strong hardware relation in the embedded domain, the costs for hardware have a great influence on the overall system costs including hardware, software, production, and installation costs. Thus, controller costs should be given as property of a controller to be able to estimate the hardware costs of the system.

Actually, the costs may change not just over time but even over the number of units taken from the vendor.

2.4.2 Sensors and Actuators

Sensors and actuators provide the embedded system interface from/to its environment. They provide and require signals and thus need to be connected to communication lines for propagation of the respective information. Because of the different types of sensors and actuators to be used, their connection can be quite different. The differences are handled by the device-to-commline connector.

Although signals are usually not considered as hardware, they are considered as components in the hardware architecture to build the interface between hardware, i.e. the sensors and actuators, and software, i.e. the functions containing signals. An alternative in order to avoid sensors and actuators to directly provide and require signals, pseudo functions might be mapped to them instead (cf. AUTOSAR, see Heinecke et al. [HSF+04]). The meaning of functions as part of the embedded system's software-based functionality would be extended to functionality of hardware. To avoid mixing up hardware and software, the actual interface between them has been chosen to be based on signals and not on pseudo functions. Thus, signals are considered as components in the hardware architecture. Furthermore, the actually existing abstract character of signals as information to be communicated is covered by this modeling decision because no distinction between the providing and requiring component types is made by a signal itself.

At least when looking to particular parts of an embedded system, not all information required or provided will be available or consumed inside the system or by the interface based on sensors and actuators. Often, interfaces are realized by buses propagating the respective information. Section 2.6.3 addresses this kind of interface of subsystems.

Again, a non-technical but important property of sensors and actuators is their costs. Like controllers, the great number of units in the embedded system domain justifies a thoughtful consideration of even small cost differences between variants of a single device. Furthermore, the documentation of costs for understandability of architectural decisions still has a great meaning. With costs as driving artifact, the shared use of sensors especially their provided signals is motivated. Moreover, installation space can be saved which is not just short but the position of a sensor often is predefined by its field of application. The installation of several sensors may be impossible not just because of the costs but because of the exclusive positioning requirements. For example, the rotation wheel sensor—it is a physical part of the brakes—provides the velocity signal required by a great share of the overall functionality. Obviously, an exclusive sensor for every function (e.g. antiblock system, cruise control, air conditioning, radio, locking system, etc.) is not practicable. Although

a redundant installation of many sensors is available for safety reasons, they do not provide the same signal to different functions.

2.4.3 Communication Lines

Communication lines are used to propagate information throughout the system. They are linked to devices but are no connectors. The reason for this is based on the different types of communication lines which need to be connected in different ways to devices. Thus, additional connectors become necessary to describe those connections. Communication lines can be just lines to carry information as pulse or even as power to drive an actuator. Extended communication lines are e.g. buses. Actually, buses are pulse-driven lines as well but with another level of interpretation. This differentiation is addressed in Section 2.6. In this section, communication lines are presented in terms of connectivity in the hardware architecture. Hence, they are hardware components with mostly physical background.

As one important property, their diameter needs to be sufficient to conduct the power necessary for communication or even driving an actuator. In combination with the line length, the communication line weight and costs can be expressed. Besides the assignment of the lines (see Section 2.4.6), the weight to be carried by the car and especially the costs are most interesting properties of the communication lines in evaluation. A complete cable harness of an automotive system might weight more than one hundred kilograms. More weight leads to an increased fuel consumption. Additionally, the costs for the raw material, the copper, increase the overall costs of the final product. Like devices, communication lines should be selected under thoughtful and economical considerations.

2.4.4 Device-to-commline Connector

As shown in Figure 2.3, connectors (denoted with "2") are located between a device and a communication line. Again, there are no connectors between two components of the same subtype.

Usually, there is an electrical connection from a controller to e.g. a sensor, represented by a communication line, on which voltage is applied. In case of simple communication lines, the connector describes how to represent the abstract communication on the line. This is usually done by applied voltage. For example, a rotary door latch can determine whether a door is opened or closed. An opened door may be represented as applied voltage (e.g. 5 volts). A closed door is represented as no applied voltage.

Moreover, it is possible that some information is not directly available as a signal. Let the sensor be a hall sensor which counts rotations of an electric motor. This sensor does not provide the absolute position of the window powered by the motor but its relative one with respect to its old position. Such a complex interpretation of

a signal has to be realized by an additional software component that computes the absolute position and is not to be handled by a device-to-commline connector.

A special connection may be that from a controller to an actuator. An actuator needs to be driven by power. Thus, in most cases, the communication line conducts power to the actuator simply switched or even controlled by the controller. In those cases, the connector has to represent more than a one bit connection in terms of applied or no applied voltage. A more detailed one with several steps or even a stepless one becomes necessary. An example can be given with smooth window closing. When the window approaches its final position (e.g. closed), the electric motor has to be slowed down to achieve the smooth closing. In this case, a simple on/off control is not sufficient. As a consequence, more than a one bit information has to be communicated via the line.

In case of a bus to be connected to a device, the connector has to take the bus protocol to be used into account. A respective transceiver has to be physically available at the connected device. Usually, just controllers are connected to buses. The connection of sensors and actuators has to be realized via more or less simple communication lines. Nevertheless, some sensor and actuator types are available as so-called smart sensors and actuators, respectively. In this case, they are able to interpret, receive, and send bus messages on their own. This provides a more decentralized positioning with considerably higher costs as trade off.

2.4.5 Omission of Power Supply

The power consumption of devices is another important property of an embedded system. Actually, the controllers and actuators are the components that consume most of the available power. Nevertheless, the power consumption of controllers cannot be initially given independently from their application in the embedded system. Their need for power strongly depends on the functionality realized by the embedded system and thus has to be evaluated after the system architecture has been built. To supply especially distributed systems with power, additional power supply lines have to be added. In the embedded system architecture presented in this chapter, this power supply is omitted for now. The intention of the architecture model is the representation of architectural decisions. Because of the distribution of power consumers over the whole system, i.e. the installation space, power supply can be taken as granted for the moment. Actually, it has to be designed for a particular architecture. Thus, the architecture is input for the power supply design and not vice versa. The differences in power supply layout can be enormous for one and the same system (cf. Gemmerich et al. [GSZ+05]). Nevertheless, those differences can hardly be taken into account in early stages of system architecture. Moreover, differences in the architecture may not cause significant differences in the power supply, because most of the consumers have to be placed invariant with respect to architectural decisions. Their location regarding the installation space is predefined by their intention.

Especially actuators have to be installed in places in which their activity is needed. For example, break lights have to be installed at the rear end of the car and power window motors near the respective window.

The concern of the layout of the system, or more precisely of the network, is addressed in Section 2.4.6. Although it is not considered as part of the architecture in terms of representing architectural decisions, its meaning for the overall system is outlined in that section.

Although power supply may be available at nearly every geometric location in the system, there is one problem that is taken into account already in the architecture development by the hardware architecture. The approximate length of communication lines can be taken as basis for estimations of the costs and the installation problems of the cable harness (see Section 2.4.6). Moreover, a direct powering of actuators over a relatively long distance leads to long communication lines. Those need a particular diameter to actually conduct the needed power. Such power providing communication lines cause additional costs and weight to be considered in the architecture development.

2.4.6 Network Layout

The alignment of the cable harness containing communication and power supply lines is one part of the layout. Positioning of controllers, sensors, and actuators is another one. Besides purely geometric restrictions, environmental conditions have to be taken care of. The network layout may be considered as subdiscipline of hardware architecture. Because of its complexity, it is not addressed by that view on architecture but separately mentioned.

Actually, a three-dimensional geometric model is necessary to exactly calculate the alignment of the cable harness. But such models are rarely available at the architecture development level. Because of strongly restricted geometric options, the alignment of the cable harness cannot be done arbitrarily anyway. The predefined alignments will support estimations of the length of communication and power lines if the locations they have to reach are approximately known. These locations of sensors, controllers, and actuators, which are to be connected via the cable harness, can be modeled by properties of the devices. A partitioning of the installation space in predefined locations can be done on the architecture level already. Thus, locations like e.g. driver door, engine compartment, rear end left side, and so on are sufficient to estimate the length of communication and power lines. Additionally, parting points[1], which bear hardware and installation costs, can be taken into account in between certain locations. Thus, the costs of the cable harness based on its alignment and preset by the positioning of devices can be evaluated even without exact

[1] Parting points are special plugs at which the cable harness is separated initially. Usually, the harness is not parted any more after installation at this points.

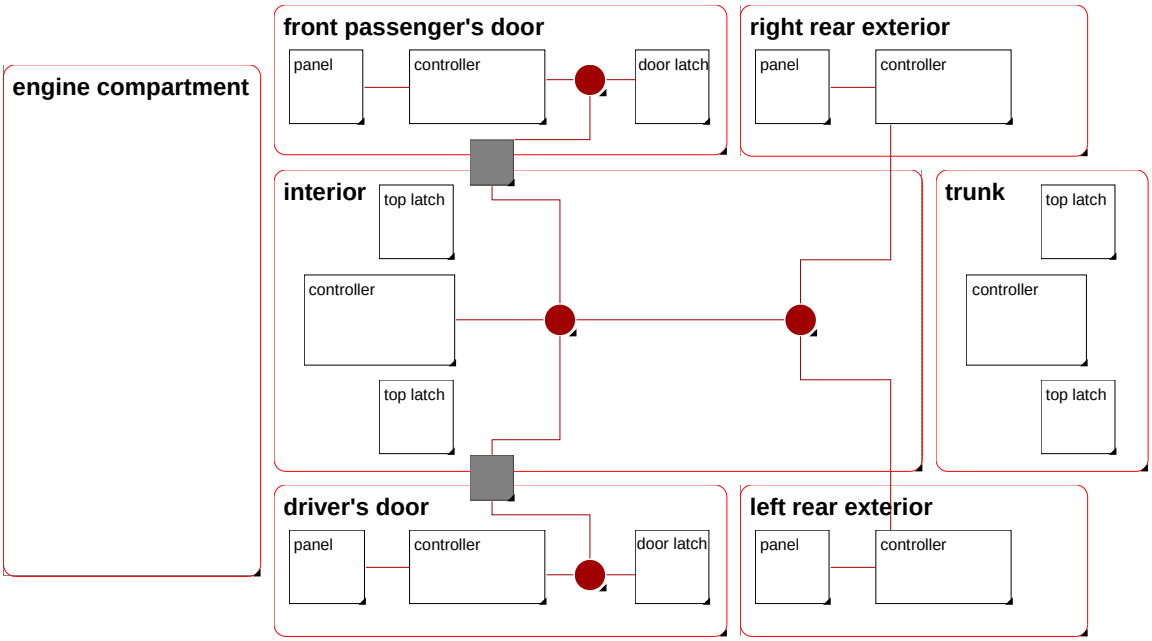

Figure 2.7: PREEvision topology model

three-dimensional models. An example is given by a so-called topology model of the architecture tool PREEvision (cf. Ringler et al. [RSB07]) as can be seen in Figure 2.7. By supporting information on the length of the predefined cable conduits, the length of any communication line aligned in the conduits can be estimated.

Nevertheless, after the architecture and thus its network layout have been determined, there is still much potential in the cable harness design. A detailed modeling including parting points, fuses, branches, and plugs allows optimization regarding hardware and installation costs of the harness. While coarse-grained considerations of these elements are implicitly made by the designer of the network, a high-precision integration of this problem needs to be taken into account in separate. In Gemmerich et al. [GSZ+05], an application of layout algorithms is presented to achieve a cost-efficient layout of the cable harness. Because of the complexity of this problem, an integration in early architecture may lead to high modeling and evaluation efforts. Thus, the application should be deferred to later phases of development.

2.4.7 Types and Instances

Like in the function architecture, the components of this architecture view should be kept as types to be instantiated in later application. Representing real hardware components instead of virtual software components, their instantiation matches their actual installation. This points up the fact that instantiation is no further extension

but simple application with assigning names. Nevertheless, defining types is important for the same reasons as already discussed in the types and instances section of the function architecture (see Section 2.3).

Again and more obvious, the choice whether component or connector is supertype of an architecture element can be confirmed by the argument that components can exist on their own while connectors are not viable without components. Actually, an instance of a connector can exist without context but does not make any sense. Thus, elements that can actually be touched by some installer are components, the connection of those components represented by connector instances are created by the installer. Communication lines are components that are often considered as connectors because a connection is built via this type of component. With respect to the previous argumentation, communication lines are considered as components of the hardware architecture.

Signals are components of the hardware architecture for interface reasons. Nevertheless, they should be handled in the same way as in the function architecture. For modeling purposes, there is no significant difference in signal type definition or instantiation regarding their application in function or hardware architecture.

2.5 The Function Mapping

The function mapping connects components of the function architecture, the functions, to components of the hardware architecture, the controllers. Hence, this view of the embedded system architecture has a connecting (say mapping) intention. The components of this architecture are shared components, no new ones are introduced at this point. With respect to a function-oriented architecture development in contrast to a controller-oriented one, the mapping of functions is of particular interest as it provides enhanced flexibility. Figure 2.8 depicts the metamodel of the function mapping. Its essential elements are summarized in Table 2.3. A more detailed description of the elements, the construction, and the application of the metamodel is given below.

components	
functions	Same as in function architecture (see page 21).
controllers	Same as in hardware architecture (see page 26).
connector	
function-to-controller	A function-to-controller connector maps a function to a controller.

Table 2.3: The function mapping

2.5.1 The Components

As already mentioned, the components of the function mapping are shared with the function and the hardware architecture. A function is the smallest unit to be mapped. A feature as part of functionality consists of several functions realizing it. For example, a power window system consists at least of the a Power Window Control function for each of the windows and a coordinator function. Depending on the kind of power window system, additional or extended functions for more windows and further functionality like short lift (see case study in Section 3.1) can be part of the feature. Smaller units like tasks or processes are not yet considered because they do not represent architectural decisions which is the main intention of this embedded system architecture model. They have to be taken care of *due to* the function mapping and thus are not to be defined *by* the mapping. But at least for more detailed performance evaluation, they become quite interesting. In that case, the functions may need some refinements representing their inner structure of tasks and processes.

Nevertheless, the components relevant for the function mapping are the functions of the function architecture and the controllers of the hardware architecture. Although a more detailed view on the mapping will become necessary in the design phase, for architecture purposes this precision is sufficient.

2.5.2 Function-to-controller Connector

A function is connected (say mapped) to a controller by a function-to-controller connector. In the first place, the connector just determines which function to be mapped to which controller. In most architecture models, this mapping is represented by an allocation viewtype (see Clements et al. [CBB+02]) instead of a C&C viewtype. Hardware components play a central role in embedded systems. For this reason, they are represented as components of the architecture instead of just a hardware platform to be allocated. Hence, a connector is needed to connect function components to con-

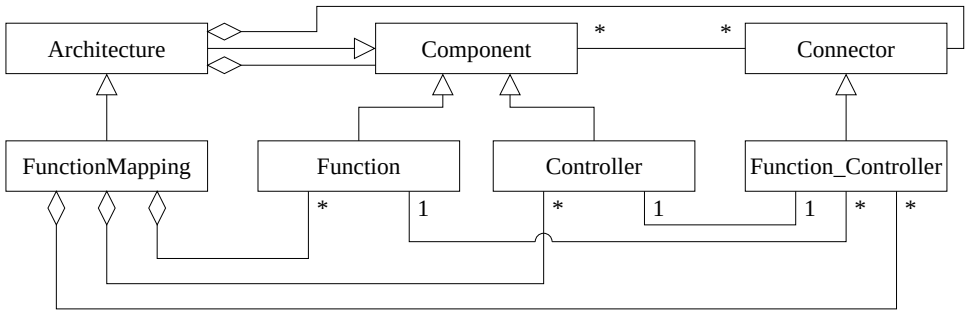

Figure 2.8: The function mapping metamodel

troller components. In the second place, further information on the configuration of controller and function can become interesting, which are not yet needed to describe the differences of architecture variants.

2.6 The Communication Mapping

In contrast to the function mapping, the communication mapping represents way more details about the actual connections. First, a signal is not necessarily connected to just one communication line. Second, the connector itself carries more detailed information on how to represent the signal on the respective line. That does not mean that the function mapping is less important. Actually, it is required by the communication mapping to determine which signals have to be communicated between hardware devices. The focus of this view on architecture is the inter device communication while the intra device communication is not addressed here. Figure 2.9 depicts the metamodel of the function mapping. Its essential elements are summarized in Table 2.4. A more detailed description of the elements, the construction, and the application of the metamodel is given below.

components	
signals	Same as in function architecture (see page 21).
communication lines	Same as in hardware architecture (see page 26).
connector	
signal-to-commline	A signal-to-commline (short for communication line) connector maps a signal to a communication line. Although signals are connected by signal-to-signal connectors, they are mapped independently to communication lines. Actually, the respective connectors have to match.

Table 2.4: The communication mapping

2.6.1 The Components

Like the function mapping, the communication mapping does not introduce any new components to the embedded system architecture model but connects shared ones (see Table 2.4). There is one significant difference how the components are shared with respect to the other views. The signals are not considered to be a part of another component like in the function and hardware architecture as part of a function, sensor, and actuator, respectively. Their affiliation is explicitly not mentioned because

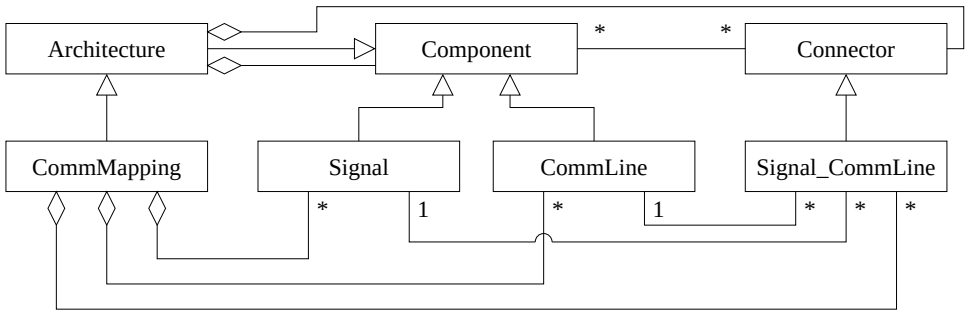

Figure 2.9: The communication mapping metamodel

it is already covered by the previous views without which a communication mapping does not make any sense in most cases. In those cases, in which a communication mapping is reasonable even without the previous views, i.e. as interface for an incomplete system or a subsystem, the affiliation is usually still unknown anyway (see Section 2.6.3). Hence, the signal affiliation is not taken into account by this view. However, the communication lines are used like in the hardware architecture.

2.6.2 Signal-to-commline Connector

The signal-to-commline connector connects a signal to a communication line. Its actual appearance differs regarding the underlying communication line. Figure 2.10 arranges the C&C-based scheme of Figure 2.3 to the Open System Interconnection (OSI) reference model (see Zimmermann [Zim80]). Although, an embedded system is typically not referred to as open system, its devices do communicate with each other. They are not closed, thus have an interface for interconnection with others. With sensors and actuators building the interface to the system environment, even communication from outside is supported. An embedded system is static in most cases. In contrast to most open communication systems in which new components can be added quite easy, the communication participants are known at design time already. This fact may let an embedded control system appear less open. Nevertheless, an arrangement to the OSI reference model can be useful to distinguish the different connectors used on the views as well as to outline the differences of various signal-to-commline connectors. Furthermore, the realization of communication principles in the embedded system can be made explicit and descriptive.

In Figure 2.10, the underlying hardware platform is outlined by the gray columns connected via the physical interconnection media (layer 0). The gray parts represent an example instance of the OSI architecture. The columns can be considered as devices of hardware on which functionality may be mapped (cf. connector “3” in Figure 2.3). In case of a sensor or actuator, the application layer can be neglected as no function can be mapped on them. The arrangement of embedded system

architecture elements to the OSI layers is discussed in the following paragraphs. The seven layers of the OSI architecture are shortly introduced at the beginning of each of the paragraphs. See Zimmermann [Zim80] for a detailed introduction to the OSI reference model.

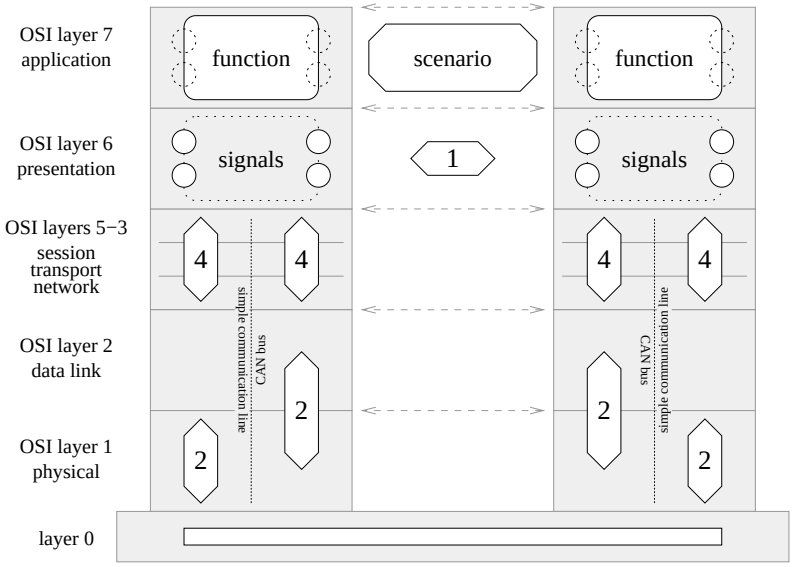

Figure 2.10: The elements of the views mapped to OSI reference model

Application Layer (7) The *Application Layer* is the hierarchically highest layer and thus contains the highest abstraction of distributed communicating application entities of the OSI architecture. This level of application is directed to the communication of the system. The actual functionality of the system is located on top of the Application Layer.

The functions of the embedded system architecture as parts of the functionality and data processing units are located at the Application Layer. To process a scenario, e.g. a use case, functions will process information required from sensors and other functions to provide to other ones as well as to actuators. The need for communication is given by a scenario represented by e.g. a sequence diagram. It contains information on which data to be processed by which functions and thus via which devices and communication lines.

Presentation Layer (6) The *Presentation Layer* provides the interpretation of the communicated data based on their presentation. The location independence of this layer, and thus platform independence, enables the direct linking of its entities without considering the underlying layers.

For a common representation of data, signals are introduced to describe required as well as provided information (see Section 2.3). Signals provide a bitwise representation of the data to be communicated. The concrete representation enables the linking of functions regarding their interface (called signature in the computer science domains) defined by signals. The signal-to-signal connector is used to link signals considering their respective meaning.

This abstract handling of signals is needed to provide a common basis for interfaces of system components and the system environment. The utilization of the OSI architecture layers refers to their technical realization. Hence, the necessary distinction of different appearances of signals is considered by the OSI architecture as presented below.

Session, Transport, and Network Layer (5-3) Although there is a strongly intended distinction between the Network, Transport, and Session Layer, they are taken into account in combination at this point. Particularities of exemplarily applied technologies are mentioned below to outline the application of the signal-to-commline connector. Following, the three layers are shortly introduced.

Functional and procedural means for network service data exchange between transport entities are intended by the *Network Layer*. Its services are based on the *Data link Layer*. The *Transport Layer* provides data transfer between session entities. Higher layers do not need to be concerned about how to actually provide reliable transfer. This is concern of this layer. The *Session Layer* assures the cooperation between entities of the Presentation Layer. Besides a binding of these entities, the control (say coordination) of the presentation entities is taken into account by this layer.

From layer 5 down to layer 1, the content of the layers depends on the technology applied. Different contents allow independent communication on the upper layers (6 and 7). Two examples are given: A simple communication line and a CAN bus. A signal-to-commline connector specifies the transition from the upper layers to the lower ones. Actually, the content of layers may be implicitly implemented by the chosen technology which will be mentioned for the examples.

Applied voltage may be used on the Physical Layer for communication and is relayed by the Data link Layer in case of a simple communication line (see below). The signal-to-commline connector, located at layers 3 to 5, maps the signal representation to the communication line. Thus, an interpretation becomes necessary to be able to link two entities of the Presentation Layer (actually signals) on the Session Layer. This is not directly done by the connector but by referencing the communication between session entities. Because of the trivial realization of simple communication line connections, neither the services of the Transport nor of the Network Layer are explicitly implemented. Routing and switching, usually handled by the Network Layer, are realized by the fixed connection to devices. Because just one sender is allowed

in this case, the communication direction is non-ambiguous. The possibilities of the Transport Layer are restricted accordingly. As a consequence, the mapping of a signal to a communication line is represented by a mapping to the lower layers which provide applied voltage as connection characteristics for simple communication lines. For instance, an applied voltage of 5 volts may be interpreted as logical `1` whereas no voltage applied may be interpreted as logical `0` or vice versa.

In case of communication via a bus, e.g. a CAN bus, the Network and the Transport Layer have more abilities. The Data link Layer already provides the frames for CAN messages. To keep costs low, especially for CAN transceivers, dynamic transport is disapproved. Predefined messages are used for simple access to the contained information. This definition can be accounted to the Transport Layer. Because of the simple hardware structure of CAN, switching and routing are not directly supported. Acceptance filtering is located on layer 2, which ensures syntactic correctness of frames. But, special CAN transceivers support filtering messages regarding their ID and content by masks. Only parts of a certain frame will be accepted and directly written to e.g. some register of the controller. That way, the transceiver does not need to support arbitrary CAN messages, which would be more expensive to realize. The content of the messages may be directly (interpretation inclusive) provided by the transceiver. Additional hardware or software for interpreting arbitrary messages can be saved as well. Moreover, addressing certain controllers can be realized in that way instead of just broadcasting messages via the bus. Although this procedure seems to be somehow inverse, it supports simple addressing from the receiver's point of view. Therefore, information on how to filter messages regarding their content is located on the Network Layer.

Data link Layer (2) Functional and procedural means to provide data links between network entities is the intention of the *Data link Layer*. It uses services provided by the Physical Layer.

In case of simple communication lines, Medium Access Control (MAC) is realized by the connection between device and communication line. There is no dynamic access intended. Logical Link Control (LLC) is neglected as well because of the strongly limited information to be communicated. Neither MAC nor LLC services are implemented for simple communication lines. Hence, the Data link Layer can be considered as simple relay.

MAC and LLC provide the following services for the CAN bus. *MAC*: Data encapsulation/decapsulation, frame coding (stuffing), medium access management, error detection, error signaling, acknowledgment. *LLC*: Acceptance filtering, overload notification, recovery management. In the automotive domain, MAC and LLC are implemented by a CAN transceiver.

Physical Layer (1) The *Physical Layer* represents characteristics of physical connections. These can be e.g. electrical or mechanical characteristics. The characteristics have to be supported by the actual physical medium for interconnection (sometimes informally referred to as layer 0).

The connection characteristics for simple communication lines are applied voltage in most cases. For example, 0 and 5 volts may be applied to communicate between entities of this layer. For actuators, a communication line may need to provide power for driving the actuator. Besides an adequate connection and sufficient diameter of the physical medium, even special transceivers might become necessary to control the communication by power transfer.

In case of a CAN bus (at least low speed CAN), the physical medium is nothing else than a wire. Applied voltage is used on this layer like for simple communication lines. The device-to-commline connector spread across layer 1 and 2 in Figure 2.10 describes the connection via a CAN transceiver. The application and detection of voltage is handled by this transceiver.

2.6.3 Subsystem Interface

Architectural decisions often affect just parts of a system, i.e. a subsystem. To avoid modeling effort, the architecture of just the subsystem is modeled and an interface to the system and its environment becomes necessary. This interface is stated by communication. Hence, signals are required or provided across subsystem boundaries. In most cases, these signals are to be transmitted via a bus. The communication mapping is syntactically independent from the mapping between signals. Actually, the logical relations of signals represented by a signal-to-signal connector are taken into account. The advantage of the independent communication mapping is the ability to map communication, i.e. signals, even if the communicating parties are not yet known, i.e. the interaction partner of the environment is not in the viewpoint of the subsystem. Because of predefined communication on e.g. a bus, the interface to/from the system environment is contained in already defined bus messages representing sets of signals among other information. The signal-to-commline connector maps a signal to a communication line and therefore contains information on the signal representation on this line. In case of a bus, the message identifier and the positions of the relevant bits constitute the signal-to-commline connector. And in some cases, additional information on particularities regarding the CAN transceiver applied to handle the messages are added. If the subsystem is extended or even integrated, signal-to-signal connectors may be applied and a mapping of the newly added signals will have to be done according to the existing mapping of the signals of the original subsystem. If the integration is done with an existing (sub-) system, a mapping of signals to communication lines may already exist on both sides. Thus, a conformance check needs to be performed according to the signal-to-signal mappings, i.e. the logical mapping of signals.

2.6.4 Communication Mapping Alternatives

The intention of the mapping is outlined especially with respect to the OSI reference model. The legitimate question arises, why the signal-to-communication connector has been defined the way it has been. A connector between signal-to-signal connector ("1“ on the Presentation Layer) and communication line could have been defined to express the same information. The actual decision is based on several reasons exceeding the intention of simple representation of architectural decisions.

With the application of the C&C viewtype as metamodel for the different views on the embedded system architecture, a determination of what will be a component and what will be a connector has been made (see Section 2.2). A connector between a communication line and what is now the signal-to-signal connector would require the latter to be defined as component in the communication mapping. Another possibility would be a further subdivision of components and connectors in the function architecture by replacing the signal-to-signal connector by a connector-component-connector construction. The first way is in breach with the idea of keeping components as components in all views. The second way leads to a too fine-grained structure containing either dummy connectors or a dummy component. The connection of signals to communication lines is more consequent on the one hand and represents the actual realization process on the other hand.

The signal-to-signal connector represents the connection between two signals. Dependent on the communication participant, this connector may not be uniform although the provided signal is the same. The distinction between how a signal is provided and how it is actually required should not be represented in a connection between signal-to-signal connector and communication lines. Exactly the same provided signal would be mapped multiple times because of multiple and even different signal-to-signal connectors needed. This is against the intuition of providing a signal just once (ignoring redundancy for safety reasons at the moment).

For modeling purposes, it can be quite interesting to know which provided signals are communicated via some communication line already. This in combination with the option of providing the signal to multiple recipients favors a single mapping of one signal to one communication line instead of a multiple mapping of the connection between two signals. Furthermore, predefinition of e.g. bus messages is supported this way. Signals can be assigned to the content of a message even before being connected to another signal. Although it sounds confusing to propagate a signal without even knowing if and by whom it is required, predefined bus configuration containing predefined bus messages are common practice. And building systems not top down but for example based on legacy parts may require this kind of predefining communication even though not all participants are already/initially known.

To meet the common intuition of communication, fundamentally influenced by the standardization of the OSI reference model, the signal-to-commline connector is vertically arranged between entities of adjacent layers belonging to the same commu-

nication entity, i.e. the function. A direct connection from signal-to-signal connector to a communication line would mean a vertical connection in between two different communication participants which is not intended by the OSI reference model.

2.7 Architecture View Graphs

The architecture view graphs presented in this section are introduced to display architecture variants. The distinction between different architecture variants is the main interest. Many details will be omitted by the graphical representation because they are not necessary to represent a variant and thus may cause confusion. Actually, a complete architecture of an embedded system in the automobile industry has several hundreds of thousands of elements. The types of elements contained in the view graphs are given in Figure 2.11. Actually, all the elements are components. Connectors will be represented by combinations of components in most cases. This is in contrast to the common interpretation of boxes and lines view graphs that maps components to boxes and connectors to lines. The best example are communication lines and buses. Of course, they are meant to connect controllers after all. But they are components and not connectors (see Section 2.4). Trying to be compliant to the common interpretation would add nothing to the expressiveness of the view graphs but would make them big and confusing. Thus, a connection is visualized by components in touch in most cases. Figures 2.12, 2.13, and 2.14 contain abstract modeling examples supporting this argument.

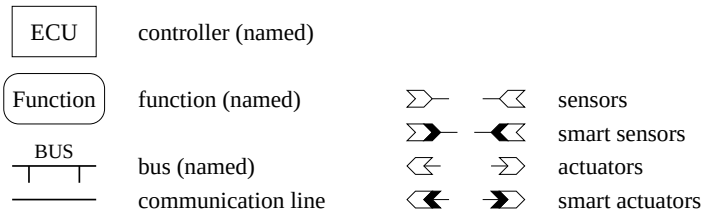

Figure 2.11: View graph legend

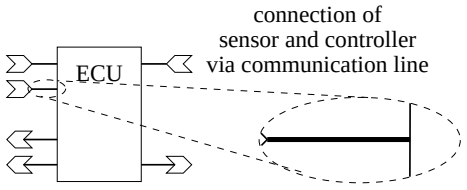

Figure 2.12: Connections between controller and sensors/actuators

Figure 2.12 depicts the connection between controllers and sensors or actuators, respectively. This connection is established via a simple communication line. There are two connectors necessary, one between e.g. sensor and communication line and another one between the controller and the communication line. These are represented by the communication line in touch with the devices.

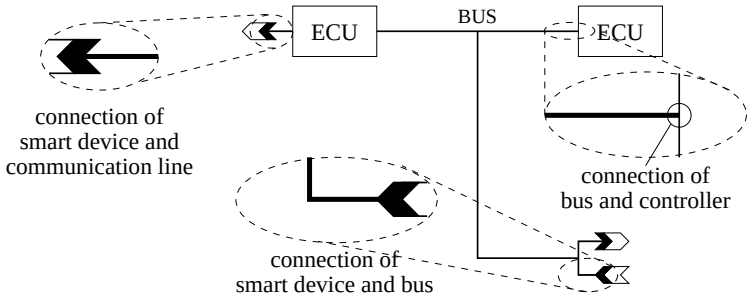

Figure 2.13: Connections to communication lines

In Figure 2.13 a connection via a bus is presented. Besides controllers, smart devices (i.e. smart sensors and actuators) are connected to the bus. The more complicated technical realization of the smart devices is expressed by their dark ending. Smart devices are not only applied at buses. To avoid long and expensive power supplying communication lines, switches can be added to actuators, which help to control the actuator via a simple communication line. This option is depicted at the upper left part of the figure.

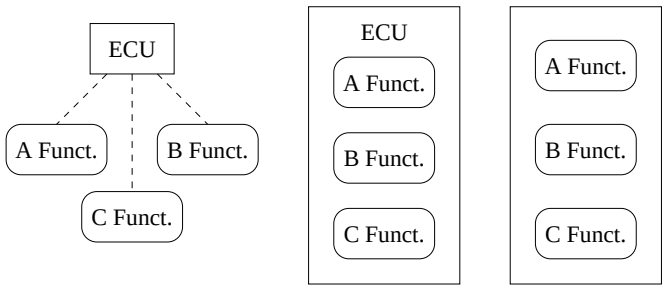

Figure 2.14: Three styles of function mapping visualization

The function mapping can be represented in different ways. The first option in Figure 2.14 contains explicit connectors (the dashed lines). If many functions are to be mapped, this type will be preferred. The middle and the right style differ just in the labeling of the controller with its name. Actually, the label does not need to contain a name. In some cases, having a significant property's value as label can be more helpful and expressive. The representation of connectors by placing the

function instances inside of a controller instance saves additional boxes and lines. This increases the clarity without omitting details.

Signals are not regarded in the view graphs for several reasons. First, even small and restricted systems can have hundreds of signals to be communicated. The representation of each of them would lead to total confusion in a view graph. Second, the connectors concerning signals are quite powerful and diverse. Thus, no common representation is available. If needed, signal connections should be separately given. Third, the communication mapping needs to be done according the communication hardware of a variant. Usually, there are no alternatives but only one possibility to communicate a signal from one device to another one. Thus, a coarse-grained communication mapping can be directly derived from the function mapping.

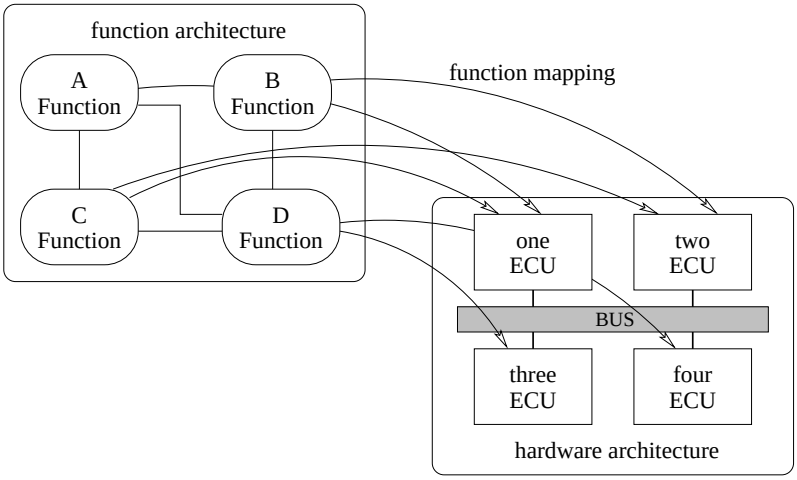

Figure 2.15: Alternative architecture view graph

An alternative to present an architecture is given in Figure 2.15. Even this small and simple system omitting sensors and actuators nearly fills up the entire drawing area. Three-dimensional (or at least ISO 3D) drawings are popular as well because of their nice appearance and extended drawing options in a layered structure. Nevertheless, the simple architecture view graphs presented above are totally sufficient to represent architecture variants and especially architecture decisions.

3 Architecture Case Studies

In this chapter, two automotive case studies are introduced by presenting their architecture variants. The first one is a Body Comfort System concerned with the power windows, the central locking system, and the convertible top developed in cooperation with the VOLKSWAGEN AG (cf. Mielke [Mie07]). The second one is a multimedia system, the In-Car Radio Navigation System, containing radio, navigation and man-machine interface functionality. The latter case study is taken from Wandeler et al. [WTVL06]. The case studies cover different states of architecture development and evaluation.

First, the development background of the case studies is different. The Body Comfort System is based on extensions of a legacy system. The extensions require reuse of (maybe changed) available components or even new development of some of them. All in all, most of the legacy system will be reused and extended, which represents the actual development process and its change from controller-oriented to function-oriented development. The In-Car Radio Navigation System is an example for variants built from scratch. The underlying hardware architecture and the mapping of functions to the devices is based on architects' decisions and without taking legacy systems into account.

Second, the Body Comfort System case study with its many variants highlights a selection-intensive architecture evaluation process whereas the In-Car Radio Navigation System with an already quite low number of variants is directed to more detailed modeling. Thus, the first one is useful to illustrate the evaluation methodology dealing with time-saving and cost-efficient reduction of the number of variants to find the most promising one. The second one—modeled in more detail—supports more reliable statements on the system quality as well as further analysis of the potential of the architecture variants.

Third, although both case studies are from the automotive domain, their different intentions and architecture rationale point out the applicability of the approach presented in this work in different fields of application.

The Body Comfort System and the In-Car Radio Navigation case studies are introduced in Section 3.1 and Section 3.2, respectively.

3.1 Body Comfort System Architecture

The Body Comfort System (BCS) case study is an embedded system for application in convertibles. A convertible is featured by a special body, i.e. its top is convertible.

The Body Comfort System contains a power window system, a central locking system, and a control function for the convertible top. Like most embedded systems in the automotive industry, new systems are built by adding functionality to legacy systems causing as few hardware changes as possible. Other than building an architecture from scratch, this policy can save cost-intensive development and takes available and established components and even architectures into account. Furthermore, common parts of several systems are much easier to support and to maintain. The Body Comfort System is based on a double extension of a Legacy Power Window Control System. The first extension is the Short Lift Control (SLC) added to the power window functionality, the second is the Convertible Top Control (CTC).

The power window functionality of the legacy system was originally built to control four power windows of a compact car. The doors of the compact car were supposed to frame the windows. Other bodies like convertibles or coupés do not provide frames for the windows by their doors. Those windows are called frameless windows. They are integrated in the doors but need to be held by the grooves at the body (especially the top) for an effective wind- and water-sealing. Thus, opening and closing the doors requires short lifts to release the windows from their grooves. The same is necessary for opening and closing the convertible top.

In this case study, the top needs to be converted manually. However, some control of the top is necessary for safety reasons. This control is concerned with providing information on the state of the top (inclusive proper locking) and with the permission of opening and closing of the top.

Two out of three architectural decisions are concerned with extensions of a legacy system regarding the power windows and the convertible top. A third decision is to be made regarding the battery to be applied in the system. It is an example for a decision that is slightly out of scope in terms of the embedded system architecture views presented in Chapter 2 but nevertheless needs to be taken into account.

In the following sections, the architecture of the legacy system as well as the extensions are presented. The latter are sectioned regarding architecture decisions, thus based on variants of the respective decision. Two variants are provided for the Short Lift Control. For the Convertible Top Control, three variants are taken into account. The battery decision provides three variants. A total of 18 variants ($2 \times 3 \times 3$) results from these three decisions as presented at the end of this section.

3.1.1 Legacy Power Window Control System

The Legacy Power Window Control System shown in Figure 3.1 is based on a Local Interconnection Network[1] with a Body Control Device (BCD) as master and

[1] A Local Interconnection Network (LIN, [LIN06]) is a low cost master-slave bus system for vehicles. It is realized by a single wire and meant to meet low communication requirements of body functionality. LIN provides transmission rates up to 20 bit/s.

four Power Window Devices (PWD) as slaves. Besides the Central Locking Control (CLC) and miscellaneous functions (misc), which are not in the center of interest here, the Body Control Device implements the Power Window Control Coordinator (PCC). The Power Window Control Coordinator coordinates the Power Window Control (PWC) instances, i.e. enabling and disabling the Power Window Controls with respect to child lock, clamp 15 (ignition), central locking, and so on. The door latch sensors are necessary for the Central Locking Control, which are connected via simple communication lines. Furthermore, a Controller Area Network[2] bus is installed. The CAN bus is out of scope for the power window system but becomes interesting for the extension by the Convertible Top Control. Actually, just a section of the entire system is shown to keep the complexity in bounds. Both, the front Power Windows Devices installed in the two doors of the body and the rear Power Window Devices installed at the left and right rear exterior installation space of the body (there are no rear doors) are all about the same. The are connected via communication lines to a power window motor as actuator, a switch, a hall sensor detecting motor rotation, and a force sensor measuring the force with which the window is powered. Additionally, the Power Window Device at the driver's door is equipped with switches for all windows and one for the child lock.

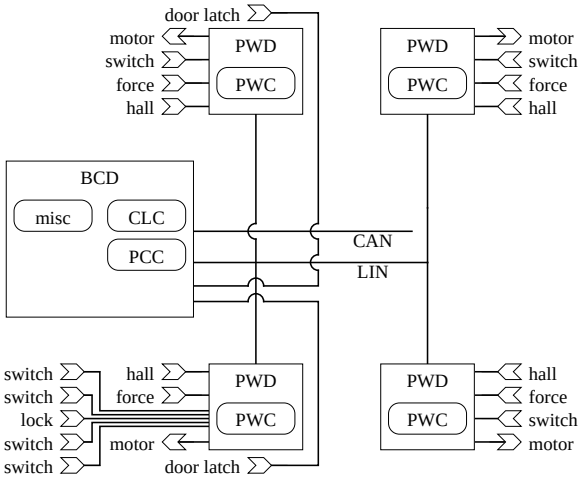

Figure 3.1: Legacy Power Window Control System

In Chapter 6, additional details on the architecture as well as on the extensions are provided with respect to the evaluation techniques applied. The descriptions of the architectures in this chapter are concerned with composition aspects and the rationale of the variants, respectively. The parts carried over from the legacy systems

[2] A Controller Area Network (CAN: Robert Bosch GmbH [Rob91]) is a bus for vehicle applications. Its twisted pair wiring enables communication which is robust against noise. Depending on the bus length, CAN provides bit rates up to 1 MBit.

are grayed out in the following figures in order to highlight the parts which are essential for the extensions.

3.1.2 Short Lift Control

As already mentioned in the introduction of this section, Short Lift Control is needed to drop and lift a power window just a little bit for releasing it from its grooves. The legacy system does not contain this function. There are two variants of extension. The first maps the Short Lift Control to the Body Control Device as it still has some resources left. The second one maps the function to the front Power Window Devices. Usually, convertibles just have two doors. There is no need for Short Lift Control for the rear windows with respect to the opening and closing of doors. The short lift required for opening and closing of the top can be integrated with the Convertible Top Control.

Short Lift Control on Body Control Device

There are two reasons to map the Short Lift Control to the Body Control Device. First, door latch sensors are already available at this device. They are required by the Central Locking Control to notice whether a door is opened or closed. Second, the Body Control Device has free resources for extensions. The latter may be a bit problematic as the reserves will be used for functionality extensions that are needed for specific bodies only (those with frameless windows). The Body Control Device will be used in various products. Thus, most of the units produced will not benefit from this extension which decreases the resource reserves for all body products. For a detailed discussion of this topic see Section 6.1.

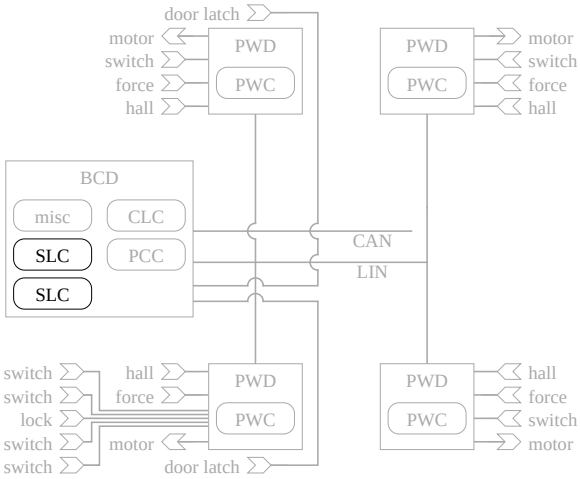

Figure 3.2: SLC on Body Control Device

Because Short Lift Control is needed for both of the windows in the front doors, two instances have to be installed. As depicted in Figure 3.2, no hardware changes are required for realizing the Short Lift Control which is an advantage of this variant of extension. A possible technical problem is given by the communication between the Short Lift Control and the respective power window motor. If the door latch detects a door is opening, the Short Lift Control has to drop the window, which has to be done immediately. The communication between the respective devices is realized via a LIN bus. The evaluation will have to assess if the relatively slow LIN bus does not cause too long delays.

Short Lift Control on Power Window Devices

The advantage of mapping the Short Lift Control to the Power Window Devices is a possible increment of performance by not using the comparatively slow LIN bus. Furthermore, the resource reserves of the Body Control Device can be saved for further extensions that do not focus systems with a special body.

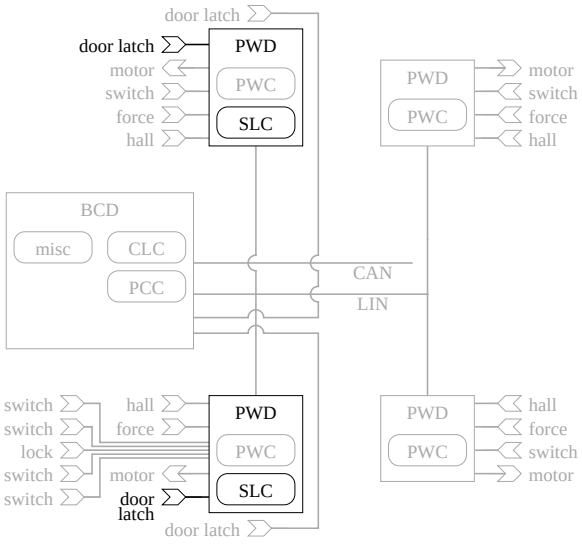

Figure 3.3: SLC on Power Window Devices

The drawback of this solution is the necessity of a hardware change as depicted in Figure 3.3 by the black Power Window Devices. Besides an extension of the front Power Window Devices, additional door latch sensors need to be installed. The already available ones cannot be connected to the Power Window Devices for technical reasons. Thus, additional hardware costs arise regardless of the costs for extending the front Power Window Devices. Although the Power Window Devices are alike regarding their technical configuration, the front Power Window Devices are not

exactly the same as the rear ones. The reason for this is the installation space available at the respective locations which requires slightly differing devices. Therefore, the front and rear Power Window Devices are to be differently built anyway, which reduces the drawback of changing the front Power Window Devices.

3.1.3 Convertible Top Control

The Convertible Top Control for observing the top state also contains some short lift functionality regarding the rear windows. Because this part of the short lift is exclusively used by the Convertible Top Control, it is not separately depicted in the architecture view graphs. The Convertible Top Control in this case study is not used to drive the convertible top. If a powered top is to be installed, the Convertible Top Control will have to be modified. There are four top latch sensors installed to detect proper locking of the top. Two are installed at the front of the top and two are installed at the rear end. They are needed by all of the variants of the Convertible Top Control. Thus, they have to be installed anyway but their connection is an interesting part with respect to their distribution throughout the body. If a controller for the Convertible Top Control is applied, its position will be the installation space at the rear end of the body next to the actuator (the top) even though it is not considered to be the powered variant in this case study.

Convertible Top Control on Body Control Device

The reason for mapping the Convertible Top Control to the Body Control Device is twofold. First, the top is part of the body. Thus, the Body Control Device is predestinated to realize its control. Second, required resource reserves are available. Their usage may avoid additional hardware installations. Actually, there are fewer bodies needing Convertible Top Control than Short Lift Control. The latter is also needed for coupés. Hence, the argument against resource usage for just a subset of the Body Control Device users arises like for Short Lift Control.

The connection of the front top latch sensors is unproblematic because of their location nearby the Body Control Device as can be seen in Figure 3.4. The top latch sensors at the rear end have to be connected to the Body Control Device via long communication lines. Because they do not need to provide any power, their diameter and therefore costs are kept low. For later modifications to support powered tops, several additional communication lines providing power for the actuators of the top need to be installed, which may drastically increase the costs. However, the manual top does not need those actuators, which makes the mapping of the Convertible Top Control to the Body Control Device worthwhile to be taken into account.

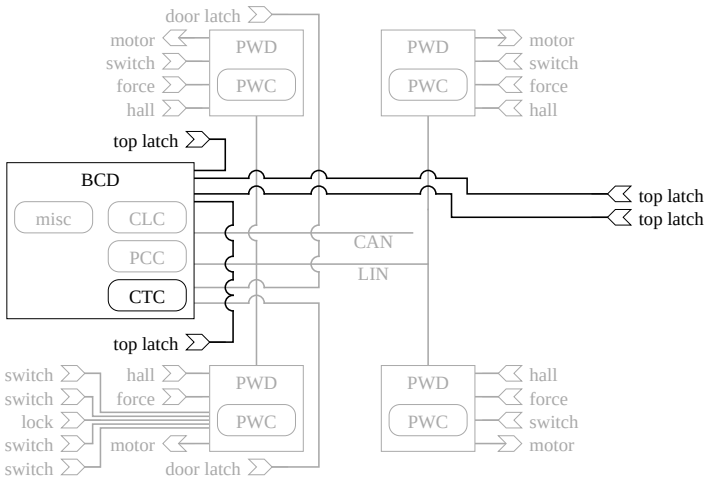

Figure 3.4: CTC on Body Control Device

Convertible Top Control on a Reduced Convertible Top Device

Convertible top devices for powered tops are already available. Because of the manual handling of the top in this case study, these devices are too powerful and thus too expensive. The reuse of legacy components, even though they need to be modified, is common in the automotive industry. Thus, a Reduced Convertible Top Device (RedCTD) is taken into account in this variant as depicted in Figure 3.5.

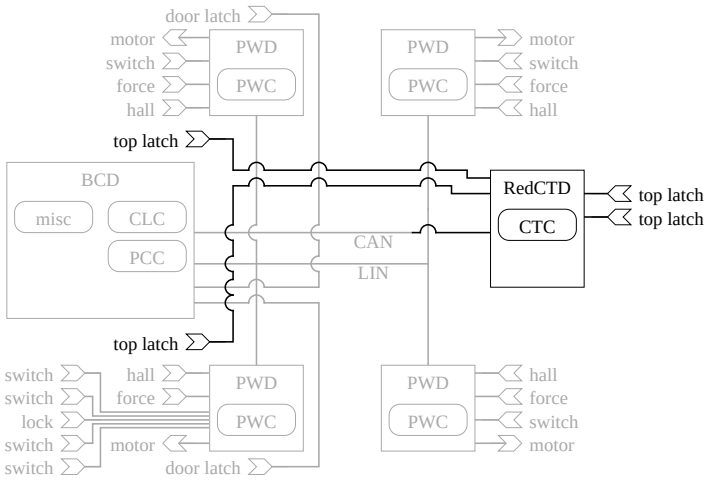

Figure 3.5: CTC on Reduced Convertible Top Device

Besides the reduction of a convertible top device, the CAN bus already installed in the legacy system needs to be extended to reach the rear end of the body in which the

Reduced Convertible Top Device will be installed. The rear top latch sensors do not require long communication lines to the Body Control Device anymore because they can be connected to the Reduced Convertible Top Device. But the front top latch sensors do need long communication lines to be connected to the Reduced Convertible Top Device in order to avoid hardware modifications on the Body Control Device. Hence, the mapping of the Convertible Top Control seems to be irrelevant for the connection of the top latch sensors in this case. But for later modifications of the Convertible Top Control for powered tops, the application of a separate device seems promising.

Convertible Top Control on Mini Convertible Top Device

As cost-efficient alternative to the Reduced Convertible Top Device, a Mini Convertible Top Device (MiniCTD) is applied in this variant (see Figure 3.6). If this variant is be realized, a modification for supporting a powered top will be more complicated than in the Reduced Convertible Top Device variant.

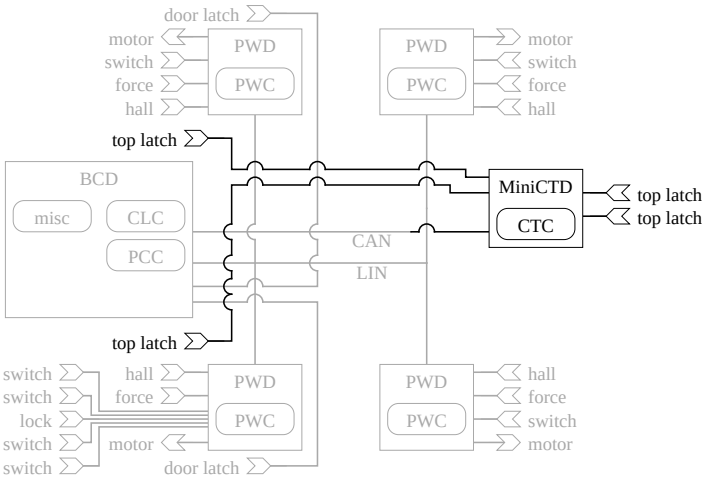

Figure 3.6: CTC on Mini Convertible Top Device

The connection of the Mini Convertible Top Device and the reduced one are alike. The main difference of this variants is based on the hardware costs of the respective device. A further difference is the modifiability to possible application for powered tops.

Variants not Taken into Account

There are two architectural decisions for which no variations are taken into account. These decisions are concerned with the connection of sensors. Due to the policy of

avoiding hardware changes, the sensors are not connected to the nearest controller available but are connected via communication lines to the nearest one that is already affected by hardware changes. Thus, none of the actual not affected controllers need to be changed regarding their hardware (i.e. the sensor connection).

The first decision is concerned with the connection of the front top latches. Dependent on the underlying variants of the Convertible Top Control, the sensors will be connected either to the Body Control Device if the Convertible Top Control is realized on it or to the separate convertible top device in the other case. In the latter, the front top latches are connected to that device as well. A variant with Convertible Top Control on a separate controller and the front top latches connected to the Body Control Device in order to relay the signals via the CAN bus are not considered. This variant would intend a hardware change of the alternatively unchanged Body Control Device.

The second decision is concerned with the connection of the additional power window switch. This switch simultaneously affects all windows and is available in convertibles only. Its location at the driver's door panel motivates its connection to the Power Window Device of the driver's door. However, if no hardware changes of that Power Window Device are required by other decisions, the switch will be connected to the controller realizing the Convertible Top Control. Thus, the controller not affected by hardware changes in order to connect the top latches stays unchanged regarding its hardware.

3.1.4 Battery Capacity

The choice of a battery does not directly affect the system functionality but affects the durability of the system in standby. Especially in convertibles that consume more power while in standby than bodies with closed top, the durability may become critical. The reason for this is the interior surveillance that is technically more complex if a hard top is missing. Usually, the battery is selected according to the engine to be started (the bigger the engine, the bigger the battery needs to be). With growing need for electrical power supply of embedded systems, the battery capacity becomes more and more important. Even though batteries with customized capacity are available or may even be developed, three standard types are considered in this case study. These types have 50 Ah, 61 Ah, and 72 Ah capacity, respectively.

There is no graphical representation of this architectural decision, because power distribution is not taken into account in detail. Nevertheless, long durability and life time of a battery are architectural requirements of a system especially because of their user noticeability. With respect of the decisions in this case study, the durability mostly depends on the standby current of the devices to be supplied.

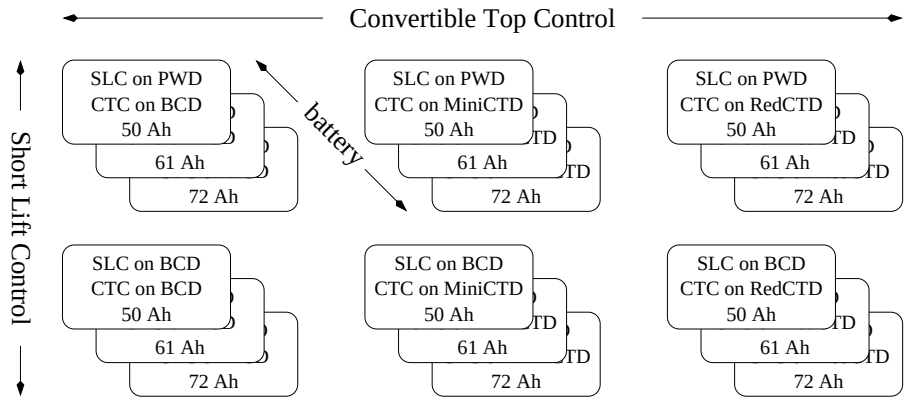

Figure 3.7: Overview of the 18 BCS variants

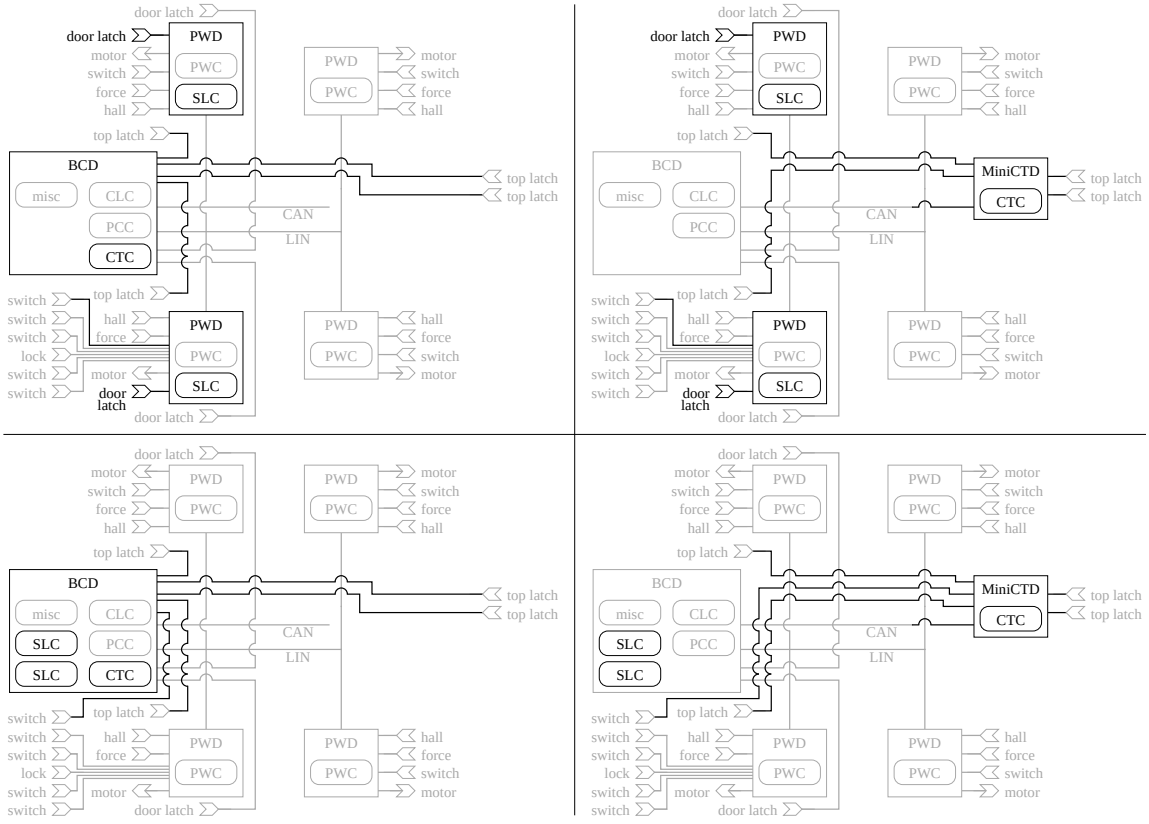

Figure 3.8: Four view graphs representing 12 variants

3.1.5 The Variants

Three architectural decisions with 2, 3, and 3 variants, respectively, lead to a total of 18 variants by building the Cartesian product as shown in Figure 3.7. The 2 variants for the Short Lift Control are vertically arranged whereas the 3 variants for the Convertible Top Control are horizontally arranged. The 3 variants of batteries to be installed are represented by the third dimension. Because the choice of the battery is not presented graphically, 6 instead of 18 view graphs can be composed. As later evaluation will show that the Reduced Convertible Top Device will not bring up workable systems. Figure 3.8 depicts the four remaining view graphs representing 12 variants with respect to the battery decisions. Table 3.1 enumerates all of the 18 variants.

variant			**in Figure 3.8**
SLC	CTC	battery	
BCD	BCD	50 Ah	down right
BCD	BCD	61 Ah	down right
BCD	BCD	72 Ah	down right
BCD	MiniCTD	50 Ah	down left
BCD	MiniCTD	61 Ah	down left
BCD	MiniCTD	72 Ah	down left
BCD	redCTC	50 Ah	no
BCD	redCTC	61 Ah	no
BCD	redCTC	72 Ah	no
PWD	BCD	50 Ah	top right
PWD	BCD	61 Ah	top right
PWD	BCD	72 Ah	top right
PWD	MiniCTD	50 Ah	top left
PWD	MiniCTD	61 Ah	top left
PWD	MiniCTD	72 Ah	top left
PWD	redCTC	50 Ah	no
PWD	redCTC	61 Ah	no
PWD	redCTC	72 Ah	no

Table 3.1: BCS variants overview

3.2 In-Car Radio Navigation System Architecture

The In-Car Radio Navigation System (ICRNS) case study describes a multimedia system with functionality consisting of radio, navigation, and man-machine-interface. This case study was introduced by Wandeler et al. [WTVL06].

Besides standard radio functionality like volume control, a tuner, and so on, the radio function (RAD) contains RDS (Radio Data System) and TMC (Traffic Message Channel) handling. The latter will be required by the navigation function for replanning routes in case of traffic jams. The navigation function (NAV) consists of user input support for the destinations, route planning, and guidance by audible and visible advices. Besides interaction with the radio and man-machine functions, a connection to a database as well as relative and absolute positioning information are required which are omitted in this case study because they can be realized outside the focus the architecture modeling of this approach. The man-machine interface (MMI) is concerned with user interaction. Hence, the input key handling and the output via display are the main intentions of the man-machine interface.

3.2.1 The Variants

Figure 3.9 shows three of the originally five variants taken into account in this case study. In the original case study, Variant I is C, II is D, III is E. Variants I and II are based on a controller network whereas Variant III is based on a single controller solution. The network-based solutions contain a 72 kbps connection as the labels shows. The processor speed is denoted by the labels at the respective ECU. For brevity, the user noticeable actuators (e.g. keys) and sensors (e.g. display) are not explicitly modeled in this case study. They are integrated with the controller on which the respective function is mapped. Actually, they could have been modeled as well but their application is invariant and thus not in the center of interest for building variants. In contrast to the Body Comfort System case study, the variants are not based on extensions of a legacy system but given as possibly promising variants by system architects. Thus, a differentiation regarding variants due to extension is not feasible for this case study. Moreover, the low number of variants given does not motivate further differentiation. The variants will be evaluated in Section 6.2. Subsequently, the evaluation results will be combined with further analysis results to uncover architecture potential as shown in Section 8.2.

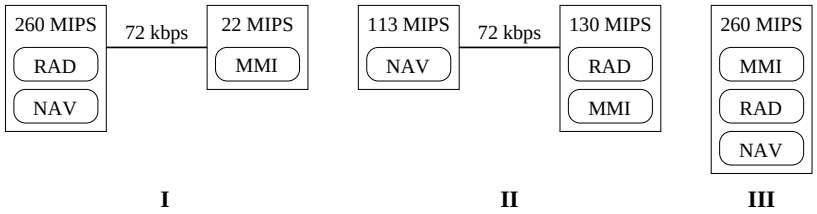

Figure 3.9: Variants of the In-Car Radio Navigation System

4 Architecture Evaluation

The requirements to be met by an architecture are represented by quality attributes. Usually, these requirements are more or less competing. Meeting all of them on the best achievable level is highly unlikely. In fact, tradeoffs have to be taken into account to achieve an overall satisfying result. Hence, the quality attributes need to be arranged in a structured way that contains a weighting of the attributes representing their relative importance. Furthermore, a detailed description of how to evaluate an architecture according to given requirements has to be provided. In Florentz and Huhn [FH06], a metamodel called Quality Attribute Directed Acyclic Graph (QADAG) has been introduced for a structured quality requirements representation. It provides a basis for evaluation methodology and even decision support. Besides knowledge of how to evaluate an architecture, information on evaluation sensitivity, effort, and soundness are quite interesting and can be attached to a QADAG instance.

Section 4.1 introduces quality attributes representing architecture quality requirements which are an essential part of architecture evaluation. A metamodel for evaluation structures, the QADAG, is introduced in Section 4.2. Section 4.3 discusses how to construct a QADAG instance for a specific evaluation. An overview of related architecture evaluation structures is given in Section 4.4.

4.1 Architecture Quality

Quality attributes represent architecture quality requirements. Besides a requirement itself, additional information on how to understand the requirement and evaluate regarding it has to be provided. In this section, an overview of architecture quality requirements is given. At first, general quality attributes as used in software architecture are presented. Afterwards, typical quality attributes as used for evaluation of embedded systems in the automotive domain are discussed.

> [...] a requirement is a system's externally observable characteristic [...]
>
> *Alan M. Davis*

Quality of software and systems often is synonymously used for fulfillment of its requirements. With architecture as development discipline, the functional and structural aspects of software and systems can separately be taken into account. A distinction between functional and extra-functional requirements is used to categorize requirements regarding their intention. Functional requirements deal with *what* a

system should do whereas extra-functional requirements deal with *how well* a system does what it is meant to do (see Gilb [Gil88, Gil05]). Besides e.g. performance and reliability, typical extra-functional requirements deal with e.g. economics and even regulations by law. Another term, behavioral requirements, has been established dealing with *how* a system does what it is meant to do (see Hochmüller [Hoc99]). The strong relation of behavioral requirements to the functions' realization and thus to functional requirements motivates to consider behavior requirements as a subcategory of functional requirements. The quotation from Davis [Dav03] shows that the term *requirements* covers both, functional and extra-functional requirements as both are observable by e.g. the user of the system. If the system does not work—with respect to functional requirements—, the user will notice. If the system does not work well—regarding extra-functional requirements—, the user will notice as well, even though the system does what it is meant to do but not on a desired level of quality. To meet both, functional and extra-functional requirements, is equally important. A system's quality will not be considered sufficient if at least the functional requirements are not met. Hence a yes/no result regarding the functional requirements can be stated. With extra-functional requirements, such a quality estimation is more complicated. A system can do what it is meant to do not just well or insufficient but more or less well. This leads to the problem of defining what actually is meant by *how well*. Another problem arises with the amount of extra-functional requirements because a combination of several evaluation results becomes necessary. Both of these problems are addressed in Sections 4.2 and 4.3. Although functional including behavioral requirements are of course not to be left out in the development, extra-functional requirements are the most interesting for architecture development and for the approach presented in this work.

The term *non-functional* is explicitly avoided because nearly every requirement—or rather its fulfillment—more or less affects the functionality of the system. Hence, a distinction between requirements affecting the functionality and those not affecting it can hardly be done in a useful manner.

Extra-functional requirements are usually represented by quality attributes (cf. Bass et al. [BCK98]). Examples for common quality attributes are availability, modifiability, performance, security, testability, usability, etc. Especially for embedded systems including hardware, requirements regarding economics, e.g. costs, need to be considered. Unit costs are strongly bound to the architecture. The choice of quality attributes depends on the domain in which a software or system is developed. Even the meaning of a quality attribute may not necessarily be the same. With respect to a common interpretation of architecture qualities, a system will have a quality attribute if the requirements are met or will not have it if the requirements are not met. There is nothing in between. In practice, this binary interpretation of quality attributes may not be sufficient. Especially in the automotive domain, usually several architecture variants are developed to be compared by evaluation. In contrast to binary quality results, quantified ones can add much value to the comparison of the

results. The effect of even slightly different architecture decisions contained in the architecture variants can be observed based on quantified results. Thus, a quantification of quality is aspired which is discussed in Section 4.4. However, to specify the meaning of a quality attribute, which is not sufficiently done by simply giving it a name (see Bass et al. [BCK98], p. 281), an explicit description of the requirements is essential. A short overview of the most common attributes and their typical meaning with respect to their application domain is provided in the following.

Quality attributes are selected in order to express desired quality of a software or system. Thus, the actual choice depends on the requirements predefined by the application domain. Will the architecture be developed for one specific customer to meet its requirements or will it be developed for several users of a specific domain? Will the user be aware of the system's existence or is it embedded into another one that hides the system? These questions are useful to be envisioned to understand the meaning of quality attributes. Nevertheless, further specification stays necessary.

The ISO/IEC 9126 standard [ISO91a] defines quality attributes for software products. The products are meant to cover requirements of several potential customers with basically the same but slightly different needs. These software products, which are not individually developed for a specific customer, are called COTS (Commercials Of The Shelf). To specify and evaluate a customer's needs, the ISO/IEC 9126 quality attributes (actually called quality characteristic in [ISO91a]) can be used. Afterwards, various software products can be evaluated with respect to the quality attributes to assess their quality and finally the most appropriate product for the customer. The quality attributes suggested by ISO/IEC 9126 are listed below.

- Functionality:
  The existence of functions with specific properties
  to satisfy the customers needs.

- Reliability:
  The capability to provide performance on a certain level
  under stated conditions for a stated period of time.

- Usability:
  The effort for use of the product by an implied set of users.

- Efficiency:
  The relationship between performance and the amount
  of resources needed to provide the performance
  under stated conditions.

- Maintainability:
  The effort needed to realize specific modifications.

- Portability:
  The ability of the software to be ported
  from one platform to another one.

The COTS character of the software is reflected by the functionality attribute. For individually developed products, the functionality can be considered available whereas COTS development may differ from the actual needs of the potential customer, which are maybe stated even after the development has been finished. A good portion of the attributes above deals with effort and resource utilization. While costs are not directly taken into account, they are implicitly contained in those quality attributes. Another attribute typical for COTS is portability. During development, the realization platform is not yet known. Even changes of the platform during life time of the product are possible. Therefore, the portability needs to be considered.

Software architecture with the background of developing individual systems is addressed by Bass et al. [BCK98]. Although such systems are not final after development, they do not have the COTS character, which shifts the quality attributes application and their meaning. Bass et al. identified the following quality attributes as the six most common and important system quality attributes:

- Availability:
  The system's availability as percentage and the
  time needed for system repairing/recovering.

- Modifiability:
  The costs and time needed for changes.

- Performance:
  The response time of the system, the effected elements
  as well as effort and money needed to provide the performance.

- Security:
  The time needed for recovery after an attack.

- Testability:
  The test execution time and the coverage of the tests.

- Usability:
  The time to perform specific tasks and the number of errors
  or problems occurring.

All of the quality attributes identified by Bass et al. are at least in parts measurable in time. A strong effort orientation regarding design time as well as runtime is contained in all of them. Although costs are not directly represented, they are addressed by the effort which covers personnel and monetary resources for development and application of the system.

The quality attributes identified as most commonly used for embedded systems in the automotive domain by Florentz and Huhn [FH06] and Florentz [Flo07b] are listed below.

- Costs:
  The costs for hardware and in some cases for software,
  e.g. license costs for navigation system software.

- Performance:
  The system load, resource utilization, and action to reaction latency.

- Physics:
  Physical constraints which are more or less user observable,
  e.g. battery capacity.

- Modifiability:
  Capability of the system architecture to be modified for
  e.g. supporting product line approaches.

The reduction of costs to hardware costs is an indication for the strong relevance of hardware costs in high volume productions of embedded systems. As the hardware is already contained in the system, its costs significantly influence the product price and profitability. Taking modifiability as quality attribute into account is not yet very common in the automotive domain. Usually, increased modifiability leads to additional costs. Possible savings of development effort for building new systems or adapting legacy systems with low modifiability can be amortized quite well because of the high volume of production. Hence, advantages of modifiability will not pay off very well at least in a controller-oriented development process. This will certainly change after function-oriented development processes will have been adapted in the automotive industry and modifications of the system and its components will become more necessary. Moreover, the reuse of components in modifiable architectures leads to decreased complexity of the systems and to less component diversity to be maintained at design time as well as at runtime. This, again, will help to save costs in development and maintenance of the systems in use.

4.2 The Quality Attribute Directed Acyclic Graph

In this section, the Quality Attribute Directed Acyclic Graph (QADAG), a hierarchical evaluation structure of quality attributes, is presented (first defined in Florentz and Huhn [FH06]). The QADAG represents quality requirements of architectures and provides the basis for analyzing architecture quality (see Florentz [Flo07b, Flo07a]). The intention of the QADAG is to integrate an explicit representation of architecture

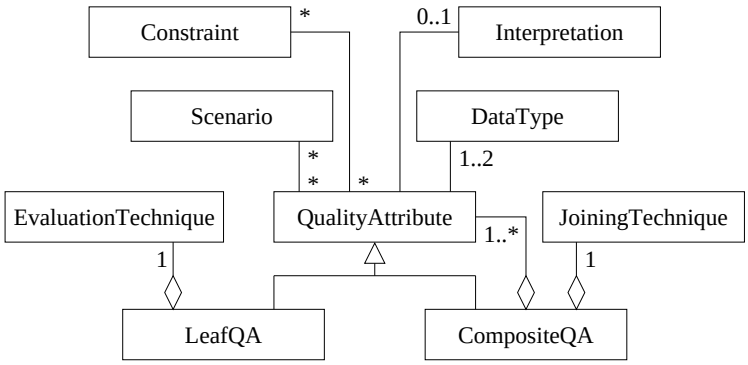

Figure 4.1: QADAG metamodel

requirements with a proper basis for analyzing evaluation results and identifying architecture potential. Figure 4.1 depicts the metamodel of the QADAG. The upper part defines the elements for describing a quality attribute in detail. The hierarchical composition of quality attributes by the composite pattern (see Gamma et al. [GHJV95]) is represented in the lower part. This structural design pattern defines a composition-based hierarchical refinement used to build the QADAG as described in Section 4.2.1. The subsequent sections will introduce the elements of the QADAG one by one.

4.2.1 Quality Attribute Hierarchy

The hierarchy of the QADAG metamodel is highlighted in Figure 4.2. Besides the commonalities of all quality attributes, an attribute is either composite or leaf. The composite pattern makes the QADAG a tree like structure, namely a DAG (directed acyclic graph). As a DAG is less restrictive than a tree, quality attributes are not restricted to occur only once within the structure. If a quality attribute needs to be considered as subattribute of more than one composite attribute, the DAG will permit such a requirement modeling. However, this special case will not be discussed in detail in this work.

Figure 4.3 depicts two different layouts of the same QADAG example instance. Which one to choose depends on the width and depth of the QADAG and documen-

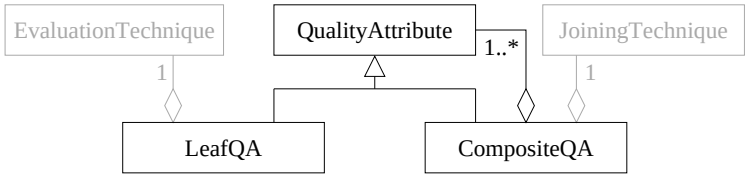

Figure 4.2: Quality attribute hierarchy (part of the metamodel)

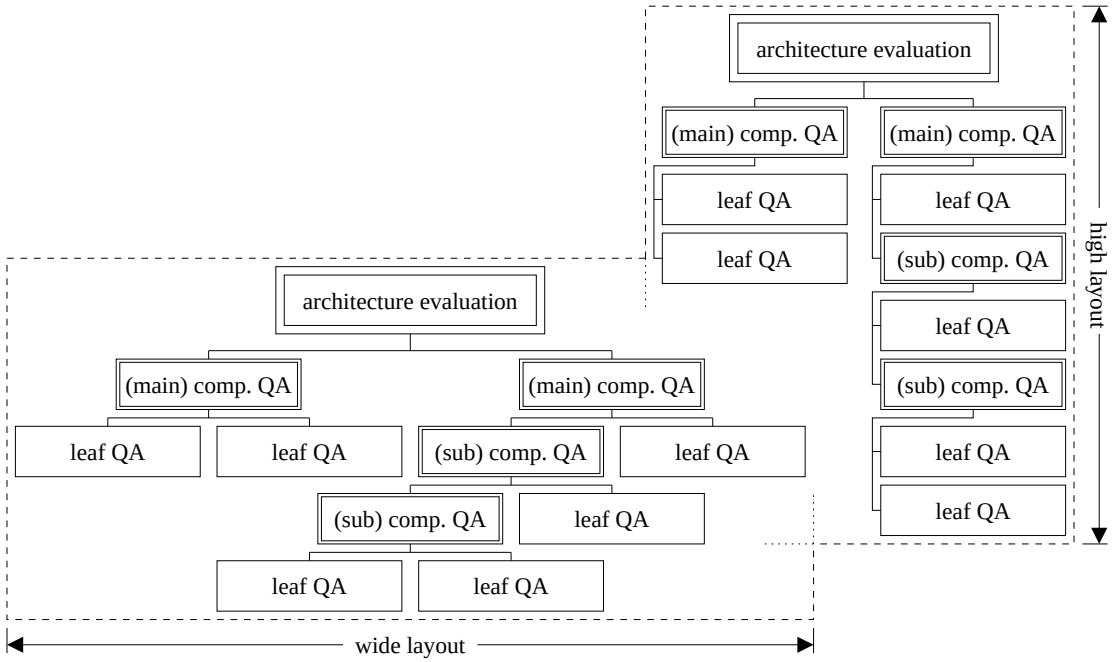

Figure 4.3: Example QADAG layouts

tation preferences. As can be seen in Figures 6.1 and 6.10, both layouts have their advantages. For instance, the first one (high layout) may be preferred for quite wide QADAG instances, i.e. with several composite attributes. The latter (wide layout) provides the separation of structural and architectural impact areas.

4.2.2 Evaluation Techniques

An evaluation technique describes how to evaluate an architecture according to a particular quality attribute. To achieve a proper description of an evaluation technique, the following information needs to be provided:

Means applied for performing the evaluation. Means refer to metrics, instrumentation, external methods, etc. and how to apply them to get the evaluation result. The choice of the means will be used to categorize the evaluation techniques according to Abowd et al. [ABC+96] as can be seen below.

Focus of the evaluation means. The evaluation means usually are applied on certain parts of the architecture. For instance, the RAM load is separately evaluated for each of the controllers. For communication performance, the communication lines are taken into account. Hence, the focus of the application of evaluation means has to be stated. In most cases, the focus is determined by the resources of components,

the evaluation means aims at. If a resource, e.g. the RAM of a controller, is not substitutable across components, these components will constitute the focus. Furthermore, only a selection of some components may be interesting for the evaluation. Again, an example is provided by communication performance. Only buses may be taken into account for communication performance whereas simple communication lines usually are omitted.

Aggregation of the subresult. In case of several subresults for the resource load of e.g. each of the controllers, an overall result has to be calculated as output of the evaluation technique. The example of resource load motivates to take the worst subresult into account as it may be a bottle-neck. Building a (maybe weighted) average of the subresults is another example. Actually, it depends on the intention of a quality attribute and has to be given as part of the evaluation technique. Modifiability is an example in which only selected parts of an architecture are to be taken into account. Specific controllers, like Convertible Top Devices, are not as important for extendability as e.g. a Body Control Device. Thus the evaluation result for the body controller will have more influence on the overall result of the evaluation technique.

Input needed to perform the evaluation. The acquisition of input data plays an essential role in the overall evaluation effort (see Section 5.1). Hence, it is quite important to denote the input data in combination with the precision necessary to avoid spending too much effort on too extensive data acquisition and bearing the risk of inaccurate or even defective output based on imprecise input.

Context of the evaluation. The context describes the situation in which the architecture has to provide the required quality. The situation is mostly given by the scenarios attached to the quality attribute that is taken into account for evaluation. Actually, the scenarios may be considered as a part of the input as well. However, scenarios are not part of the architecture modeling but of the architecture quality requirements and thus are separately taken into account (see Section 4.2.5).

Evaluation techniques can be categorized with respect to the means of evaluation applied. The categories identified by Abowd et al. [ABC+96] are *questioning techniques*, *measuring techniques*, and *hybrid techniques*.

According to Abowd et al., questioning techniques are usually based on scenarios, questionnaires, and checklists. They are also referred to as qualitative techniques. Measuring techniques are based on metrics, simulations, prototypes, and experiments. They are also referred to as quantitative techniques. As scenarios are considered to be the drivers of evaluation techniques (even measurement-based ones) and metrics are used by experts to estimate an architecture's quality with respect to scenarios, questionnaires, and checklists, most of the techniques applied in this approach can be considered as hybrid techniques. Nevertheless, the distinction between

measurement techniques and questioning techniques—the latter representing experts knowledge—is held up in order to emphasize explicit representation of measurement techniques (especially metrics) as a basis for analysis of the evaluation results and the interaction intensity of performing questioning techniques.

4.2.3 Joining Techniques

A composite attribute as hierarchy element has a joining technique attached to join the results of its subattributes with respect to a specific weighting of the subattributes. There are two ways of joining results according to their state of interpretation. Either, the results are already interpreted by interpretations of the subattributes (see Section 4.2.4). Thus, the joining can be done without interpretation but with weighting the input and normalizing the output. Alternatively and in specific cases, the results are not yet interpreted but are of the same unit and context to be e.g. added before being interpreted by the composite quality attribute's interpretation instance. An example for this case are costs of different kinds of hardware. These are separately evaluated (up to the raw value, i.e. costs in EUR) to keep the results more detailed and transparent. Costs, however, are substitutable resources. Savings at one part of an architecture can be invested in another one, i.e. the sum of costs is of interest. Thus, a combined (or joined) interpretation does make sense. However, the joining based on interpreted quality values is the commonly used one due to different evaluations, raw value units, and meanings of their result.

One additional detail of the joining techniques has to be mentioned here. The weights of the subattributes are not part of the subattributes themselves but rather part of the joining techniques. This is by reason the QADAG being a directed acyclic graph and not a tree. Hence, quality attributes may occur multiple times as already mentioned above. Their weight may be different for each of their occurrences. Making the weights part of the joining techniques avoids confusion regarding the actual weight at the respective composite attribute.

4.2.4 Interpretation of Results

The results of an evaluation are given as raw values (sometimes in terms of the response measure of a scenario), e.g. reaction latency in milliseconds. Additionally, a quality rate expression is useful, which is an interpretation of the raw value to a scale from 0 to 100 %. It denotes the architecture variant's meeting of a requirement represented by a quality attribute. While raw values are less expressive for people in an architecture development process, who are not close to a particular aspect of architecture, the interpretation to a quality rate can be seen as a common communication basis for architectural decisions.

If no evaluation technique or the necessary input data are available, expert knowledge usually will be taken as substitute. In most of these cases, an interpretation

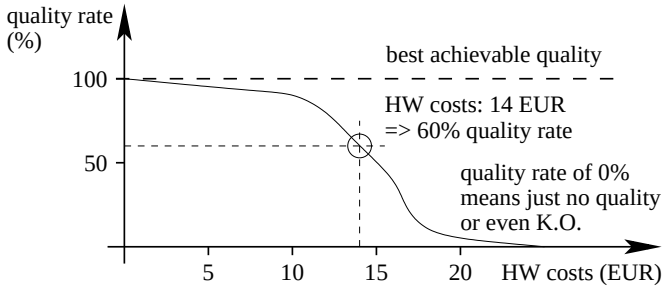

Figure 4.4: Interpretation of results

becomes obsolete because the expert states his estimation results as quality rate instead of estimated raw value to be interpreted.

The interpretation of raw values has two main objectives. First, an overall evaluation result is based on several partial results that need to be joined. Thus, a common basis is necessary which is provided by the interpretation to quality rate values. Second, the expressiveness of raw values is restricted to experts. To provide a communication platform by performing evaluation, the expressiveness needs to be improved by providing interpretations for that raw values. These will help people not familiar with the respective context to understand the meaning of an evaluation result. Moreover, the effect of changes of the raw value become transparent even though the impact of the architecture on some raw value may still be unknown. Figure 4.4 contains an example for an interpretation of costs from the In-Car Radio Navigation System case study (see Section 6.2).

A quality attribute may be declared as K.O. attribute. If an architecture is evaluated to 0 % due to a K.O. attribute, it will not be workable or even buildable, i.e. K.O. Hence, it can immediately be rejected from further evaluation and development. A 0 % interpretation regarding an attribute not declared as K.O. attribute does not lead to a K.O. of an architecture but simply means that this quality attribute is not fulfilled at all. Thus, if the evaluation result is never to exceed a specific value, this will have to be expressed by a 0% interpretation for the unacceptable values in combination with the K.O. declaration of the quality attribute.

The raw value as well as the quality rate are taken into account by the `DataType` class in Figure 4.1. The interpretation represented by the `Interpretation` class. In case of expert knowledge including an interpretation, only the quality rate will be available, i.e. one `DataType` instance and no `Interpretation` instance. In case of a composite quality attribute with a joining technique and no interpretation, the only `DataType` instance represents the resulting quality rate.

4.2.5 Scenarios

Scenarios are used to describe the context of an evaluation technique, i.e. which specific requirements has an architecture to meet in which situation. Examples for various scenarios are use cases as performance and usability scenarios and probable changes of an architecture as modifiability scenarios. As several different stakeholders require architecture quality in terms of different extra-functional requirements, the representation of the architecture with respect to the requirement differs as well. For a user, the architecture is represented by the system. Actually, as the system is not necessarily visible to the user, the architecture is represented by its functionality. A system developer view on the architecture is given by the views defined in Chapter 2. Hence, the actual appearance of a scenario depends on the stakeholder-specific appearance of the architecture. To be able to express the different kinds of scenarios, Bass et al. [BCK98] suggest six items building the basis of a scenario description.

Source of stimulus, e.g. stakeholder, developer, tester, user

Artifact of the architecture, i.e. the affected part of an architecture

Stimulus of the artifact, e.g. message, test performance, change intention

Environment of the architecture, i.e. the architectures state of development and mode of use, which is different from the environment of the system

Response to the stimulus, e.g. system reaction, development steps to perform

Response measure for quantifying the response for quality statements

Figure 4.5 shows the arrangement of the items as presented in Bass et al. [BCK98]. The response measure is most interesting for evaluation since it denotes potential evaluation techniques to be applied in a quality attribute. Architecture evaluation is used to assess the quality of an architecture with respect to the given quality attributes. These contain scenarios describing the interaction of a stakeholder with the architecture. For evaluation, these scenarios drive the evaluation technique like a test case drives the test execution. Thus, with the response measure as potential output of an evaluation technique, the scenarios can be considered to be the drivers of an evaluation technique, which provide/describe stimuli of the architecture.

As performance is one of the main quality attributes of the case studies, a performance scenario is given as an example. Figure 4.6 contains a sequence diagram describing system reactions on user interaction. It is an explicit representation of a scenario. In combination with information on the occurrence frequency of the user interaction contained in the scenario (`KeyPress()` once per second), the diagram defines the stimulus part of a scenario which is shown in Figure 4.7 in accordance to Figure 4.5. For performance scenarios considering user interaction, the user and

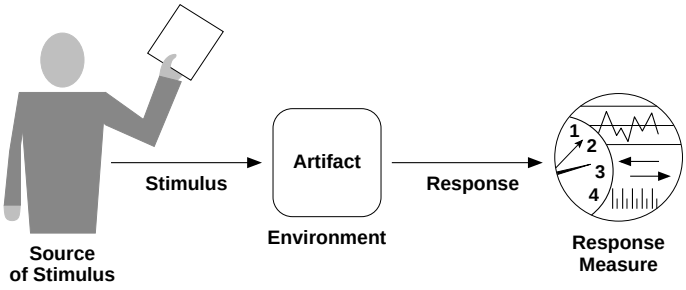

Figure 4.5: Scenario items by [BCK98]

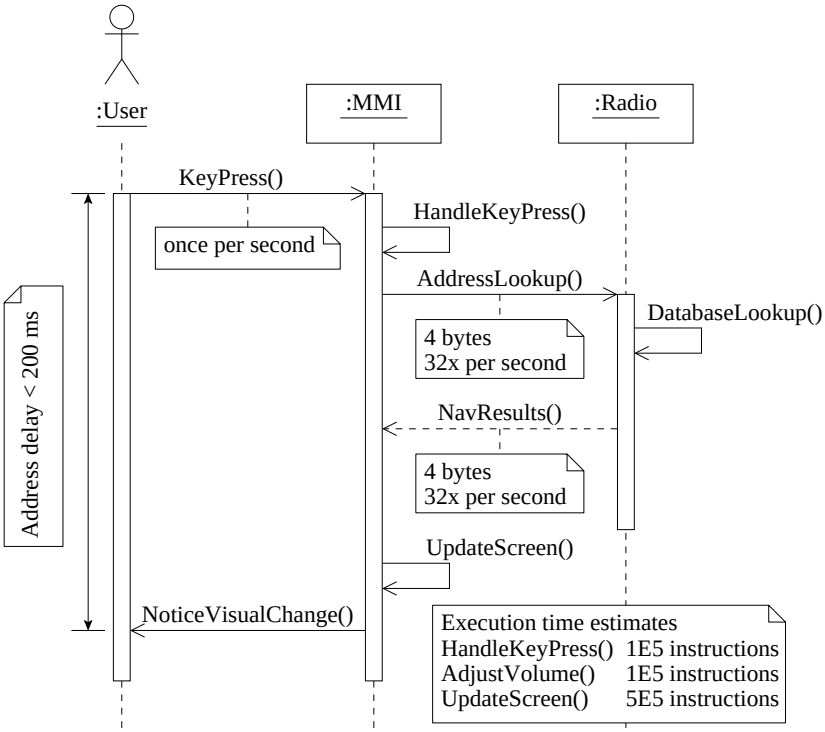

Figure 4.6: Address lookup scenario, cf. [WTVL06]

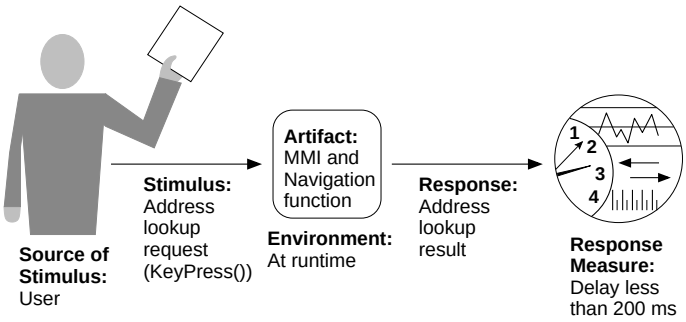

Figure 4.7: Address lookup scenario presentation according to [BCK98]

system environment behavior in terms of interaction frequency is an important piece of information to be able to calculate the system load and finally the delay of system reactions on user actions.

An example for a scenario regarding the development of an architecture is a modifiability scenario. Besides simple extensions and software updates without changing the system functionality, embedded systems are rarely modified during life time. Thus, the modifiability is directed to product line approaches and reuse in future systems.

As can be seen in Figure 4.1, several scenarios may be attached to a quality attribute and a scenario may be attached to several quality attributes as well. It depends on the type of quality attribute, which kind of scenario (e.g. performance scenario, modifiability scenario, availability scenario) may be attached. For further examples of different kinds of scenarios, see Bass et al. [BCK98].

4.2.6 Constraints

Constraints are used to set conditions to be fulfilled, that are not expressible in terms of the raw value of an evaluation technique, i.e. in terms of the response measure of the scenarios. The evaluation of performance is a good example for constraints. While the maximum latency of system reaction can be expressed in terms of the response measure (e.g. latency in ms), restrictions on the use of technology providing the resources needed for performance are to be expressed by constraints. Hence, the use or avoidance of a particular bus technology for a specific reason is a constraint because it is not possible to integrate such a requirement with the response measure. Another example is the use of hardware of selected vendors only because there are exclusive agreements. Those contracts are neither expressible as response measure nor expressible in terms of technical reasons.

Actually, checking the fulfillment of the constraints is not part of the evaluation itself. Their fulfillment can be assumed or even simply checked before the more complex evaluation process is started. Therefore, constraints are mostly simple conditions to be accepted and fulfilled during development and are not to be costly evaluated, which is the main distinction between the fulfillment of a constraint and a quality attribute. Because constraints are less strongly related to a quality attribute than its evaluation technique and scenarios, the constraint may be assigned to a composite quality attribute but still has validity for all subattributes of the composite one.

In some cases, the quality requirements of an evaluation may appear more as a constraints satisfaction problem (see Tsang [Tsa93]) than an evaluation in terms of finding the most appropriate architecture. In those cases, not the quality attribute constraints themselves are referred to as constraints of the satisfaction problem but the minimum requirements represented by the quality attributes.

As can be seen in Figure 4.1, several constraints may be attached to a quality attribute and a constraint may be attached to several quality attributes as well.

4.3 Constructing an Evaluation

A method for the construction of a QADAG—the evaluation structure—will be presented in this section. The Body Comfort System case study, which has been developed in cooperation with the VOLKSWAGEN AG (see Mielke [Mie07]) and has been introduced in Section 3.1, will be taken as an example. It covers the most interesting kinds of architecture requirements in the automotive domain. Hence, not the identification process of quality attributes but rather their hierarchical composition with respect to their usage for expressing particular requirements are in the focus of this section. The evaluation based on the QADAG instance constructed in this section will be performed in Section 6.1.

4.3.1 Hierarchical Composition

The hierarchical composition strongly depends on the specific quality attribute and the selection of its subattributes. The construction of a QADAG instance will be demonstrated on the case study QADAG of the Body Comfort System. If not already available, an assignment of quality attributes as subattributes to composite ones has to be performed. This step can be done top-down or bottom-up depending on the development process that will be accompanied by the architecture development and evaluation. Top-down approaches will be applied if superior goals are predefined but no refinement is given. Management decisions can be represented this way. The refinement has to be added by domain experts. Bottom-up approaches usually are applied when more concrete requirements in terms of e.g. scenarios are available. Those are provided by stakeholders of the architecture with special interest in one or more of the quality attributes. The scenarios can than be attached to quality attributes and some meaningful composition has to be created, which is done in stakeholder meetings and discussions. However, the affiliation of subattributes to composite ones can be taken for granted with respect to particular application domains. Hence, most approaches are neither pure top-down nor pure bottom-up approaches. Nevertheless, the case study QADAG will be built in a top-down manner because this is the more appropriate way to show and understand the meaning of the topmost quality attributes. These are performance, modifiability, physics, and costs as can be seen in Table 4.1. The table-based representation has been preferred to the graph-based one (cf. Figure 4.3) as it is capable to present weights, which will be needed later in this section. Taking physics and costs as examples, the refinement of these quality attributes will be exemplarily performed by building the attribute hierarchy.

The physical requirements (physics quality attribute) are an example for composing different as well as related subattributes. In order to avoid high fuel consumption, the weight of an automobile should be kept low. More powerful controllers do not only have more weight but rather consume more power, which has to be provided amongst

Topmost Quality Attributes			
performance	costs	physics	modifiability

Table 4.1: Topmost quality attributes of the Body Comfort System case study

others by a battery with additional capacity, again leading to more weight. Long communication lines will increase the embedded system's weight as well especially if power has to be provided by them which leads to larger cross-sectional areas of the cables. A quality attribute called weight will be used to represent the requirement of low weight. Another subattribute of physics takes an electrical property of the battery into account, that is its capacity. This property has twofold effects on the system architecture. The standby time of the system depends on the capacity available as well as the life time of the battery. But the dependencies are not the same, which leads to a further refinement of the battery quality attribute into standby time and life time. Separately considering them is quite important because they represent different requirements based on different scenarios, and after all, have different impact on the overall evaluation result.

physics		
weight	battery	
	stby	life

Table 4.2: Physics refinement in the Body Comfort System case study

An alternative to the additional hierarchy level is the direct decomposition of the physics quality attribute into weight, battery standby time, and battery life time. Nevertheless, the multi-level decomposition has been preferred in order to emphasize the relation between standby time and life time as well as to keep the separation between weight—of the overall embedded system—and the strongly battery related quality attributes.

Another example of hierarchical decomposition is the costs quality attribute. In the automotive domain, especially in high volume production, unit costs are an important driver of the development. An estimation of unit costs in the architecture phase of the overall development process is necessary to avoid too expensive architectures in terms of their realization. Software is either already contained in the controller costs (controller-oriented development paradigm) or may be—once developed—available in a digital library. Thus, in the automotive domain, software often is considered to be for free but at least its costs are covered by hardware costs. In particular in

high volume production, savings of units cost justify even high development costs because of their comparatively quick amortization. Keeping this argument in mind, the cost quality attribute is composed of cost for the ECU, the sensors and actuators, and the electrical system in this case study. Software costs are omitted for the given reason. Actually, installation costs need to be taken into account as well but are not yet available in this early stage of development.

costs		
ECU	**s+a**	**e.sys**

Table 4.3: Costs refinement in the Body Comfort System case study

After building the hierarchical decomposition like shown above for all composite quality attributes, the fundamental structure is set. Up to this stage, it is only a structured listing of quality attributes. As discussed for the physics example, a consideration of the architectural background should be aspired in order to represent the architecture requirements instead of just the organizational structure of the stakeholders. In the following section, the structure will be extended by the joining of quality attribute results with respect to their weighting.

4.3.2 Quality Attribute Result Joining

To join the results of quality attributes, a joining technique is attached to each of the composite quality attributes including the QADAG's root. To get a meaningful result for a composite attribute, the importance of its subattributes has to be expressed by a weighting. Although there is no general policy how to join the (interpreted) results of subattributes, the common technique is a normalized weighted sum. The result will be a quality rate—between 0 % and 100 %—independent of the number of results to be joined because of the normalization. The weight of each of the subattributes can be determined in different ways. The direct one is the determination of the weight in a stakeholder meeting by simply discussing their importance (cf. Bass et al. [BCK98]). Hence, determining the weighting is a more political than a technical process which can become quite obscure without sufficient insights regarding the mathematical background of the weighting. To support a more intuitive weighting, an approach called Analytic Hierarchy Process (AHP) can be applied to determine the weighting.

The AHP has been introduced by Thomas Saaty [Saa94] in 1994. It is based on a pairwise comparison of the quality attributes to determine their relative importance. This pairwise weighting is done for each of the composite quality attributes in separate. Thus, the determination of domain-specific weightings can be directed

to domain experts. The AHP is exemplarily applied for the Body Comfort System QADAG on the topmost level of hierarchy in Table 4.4.

x times more important	than perf.	costs	physics	modif.			
performance	1.00	0.67	1.33	2.00			
costs	1.50	1.00	2.00	3.00			
physics	0.75	0.50	1.00	1.50			
modifiability	0.50	0.33	0.67	1.00			
sum	3.75	2.50	5.00	7.50			
	normalized by column sum				average	impact	weight
performance	0.267	0.267	0.267	0.267	0.267	26.7 %	80
costs	0.400	0.400	0.400	0.400	0.400	40.0 %	120
physics	0.200	0.200	0.200	0.200	0.200	20.0 %	60
modifiability	0.133	0.133	0.133	0.133	0.133	13.3 %	40

Table 4.4: Analytic Hierarchy Process for the BCS case study

The upper part of the table contains the pairwise comparison of the importance between the attribute in the row and the attribute of the respective column. It is a value between 1 and 9 with 1 meaning equal importance and 9 very strong importance of the attribute in the row with respect to the attribute in the column. Values between 0 and 1 represent the reciprocal values. This part of the table can be filled during a stakeholder meeting in which questions about the pairwise importance are discussed, e.g. "How much more important is performance with respect to costs?"

The disadvantage of a pairwise weight determination is the possibility of inconsistencies. Let A, B, and C be three quality attributes to be weighted. Let A be more important than B and B be more important than C, then A must be more important than C. But in a pairwise weight determination A and C need some comparison as well. And if C is said to be more important than A, then a priority conflict will arise. Moreover, this redundancy of comparisons may lead to inconsistencies which do not affect the absolute priority of quality attributes. This inconsistency is called numerical inconsistency by Saaty [Saa94]. Let A be three times more important than B and B five times more important than C. Following A should be 15 times more important than C. Even if A was compared e.g. nine times more important than C—which is no conflict in priority—then there is still a numerical inconsistency (9 vs. 15). In contrast to priority conflicts, this type of inconsistency is not fatal but can be quantified by calculating eigen-values of the matrix containing the pairwise comparisons. Saaty and Alexander addressed the problems of conflict resolution in [SA89].

The lower part of Table 4.4 contains the calculation of the weighting. The values in the columns are normalized in order to get a sum of 1 in each of the columns. Then, the average value of each of the rows is calculated, which leads to the weight of the quality attributes. The equality of the normalized values in each of the attribute rows means that there is no inconsistency in the pairwise comparison. Little differences of the values would mean inconsistency, which is acceptable in a limited degree. However, the handling of inconsistencies and even conflicts is described in Saaty and Alexander [SA89] and is not taken into account in this case study. As already mentioned above, the weight is based on the average value. Actually, it does not matter if the weights are given as integer values or as percentages because they will be normalized during joining. Nevertheless, the percentage representation is reserved for the structural impact (see Section 5.1.1) in this approach. It denotes the absolute weight of a quality attribute regardless its position in the hierarchy. At the first level, there is no difference between (normalized) weight and impact.

Table 4.5 represents the complete QADAG instance with all its weights. The row directly above a quality attributes name contains its weight with respect of its neighbored attributes, i.e. with the same parent composite attribute. The row above the weight denotes the quality attribute's impact as absolute weight in the entire QADAG.

Weights and Impact													
26.7 %				40.0 %				20.0 %			13.3 %		
80				*120*				*60*			*40*		
performance				**costs**				**physics**			**modifiability**		
13.3 %	13.3 %			13.3 %	13.3 %	13.3 %		8.0 %	12.0 %		6.7 %	6.7 %	
100	*100*			*100*	*100*	*100*		*100*	*150*		*100*	*100*	
comm	**ECU**			**ECU**	**s+a**	**e.sys**		**weight**	**battery**		**ext**	**scal**	
	4.4 %	4.4 %	4.4 %						6.0 %	6.0 %			
	100	*100*	*100*						*100*	*100*			
	CPU	**RAM**	**ROM**						**stby**	**life**			

impact
weight
QA name

Table 4.5: Weights and impact of the BCS case study

It is important to mention that determining the weight without considering the joining technique may cause some undesirable effects. A joining technique for quality rates contains a normalization to keep the resulting quality rate between 0 % and 100 %. Because of this normalization, the impact of the joined result has to be shared by the subattribute results. This has to be taken into account when building the hierarchy and setting the weights as can be seen in Figure 4.8. Using AHP to determine the weights means taking the quality attributes into account in separate. The hierarchical refinement is not visible for the AHP. Thus, the problem with hierarchy and impact may need to be considered in separate as well.

However, if normalization is left out, which usually is the case for raw value joining like for the costs quality attribute, this problem will not appear. The specific costs

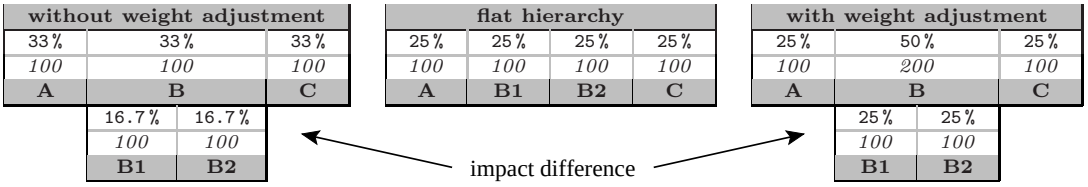

Figure 4.8: Hierarchy and impact

are added as raw values to be interpreted (after being joined) by the costs attribute. The weight and the applied joining technique have to be known to calculate the impact. Hence, it can be quite important to provide both—weights and impact—in combination as done in Table 4.5.

4.3.3 Setting up Quality Attributes

Besides a technique to evaluate an architecture with respect to the quality attribute and scenarios to describe the context of the architecture quality, the result and its use are the most important parts of the evaluation. The evaluation techniques will be presented in Chapter 6 in detail. How to handle the results in order to express architecture requirements in a hierarchical structure will be content of this section.

The first choice to be made is whether the result should be interpreted or not. Actually, this choice has already been made in the top-down approach by choosing the joining technique which either expects raw values (for later interpretation) or quality rates (interpreted values). The second choice regards the interpretation instance itself with special focus on a potential K.O. declaration of the respective quality attribute. The interpretation has to be provided by the stakeholder or an expert of this domain. Actually, the K.O. condition has to be provided too, as it is part of the interpretation. The interpretation has to be prepared to represent the K.O. For a K.O., the resulting quality rate must be 0 %. In contrast to non-K.O. attributes, this result is fatal and will not just lower the overall evaluation result but rejects the architecture variant from the development process. Thus, bad results, that are not meant to result in a K.O., need to be interpreted to a quality rate of at least 1 %. An example for such an interpretation is the battery standby time with a K.O. condition of "less than 40 days standby". Although a 40 days standby is not yet quite good either, it will be interpreted to 1 % quality rate. Nevertheless, the interpretation may even start with a higher quality for variants which just missed the K.O. condition. Again, this has to be set by the stakeholder or another domain expert.

The case study evaluation in Chapter 6 contains several examples for K.O. and non-K.O. attributes. Their particularities are discussed in the context of the respective quality attributes. Scenarios will be presented in the context of evaluation techniques and their application in that chapter as well. As scenarios are considered the drivers of

an evaluation, an isolated presentation lacks expressiveness. Moreover, the scenarios are determined by experts in most cases. Thus, a methodology for selecting scenarios for specific attributes can hardly be provided.

4.4 Related Work – Architecture Evaluation

In this section, related evaluation structures are discussed with respect to the Quality Attribute Directed Acyclic Graph presented in the current chapter. Based on the typical application domain of the evaluation structures, the comparability of the evaluation results of different architecture variants is addressed at the end of this section.

ISO/IEC 9129 Quality Models

The ISO/IEC 9126 standard [ISO91a] suggests a distinction between internal and external quality and quality in use. Internal quality is concerned with quality regarding the internal view on a software product, i.e. the quality of models, documentation, and source code. Although this quality will influence the overall product quality, it is not in the focus of quality represented by the QADAG as extra-functional requirements are not directed to the internal view. External quality in terms of the ISO/IEC 9126 is concerned with quality regarding the external view on a software product. This external view is related to the extra-functional requirements represented by the QADAG. Despite the distinction of internal and external quality, both are represented in the same hierarchical structure of so-called quality characteristics and subcharacteristics. Actually, the characteristics and subcharacteristics can be understood as quality attributes and subattributes. The topmost characteristics are *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability* as presented in Section 4.1 (see Figure 4.9).

To asses the quality regarding a characteristic, a metric has to be identified. The assessed quality has to be rated to *excellent*, *good*, *fair*, or *poor*. The distinction between internal and external quality has to be taken into account by the selection of the metrics. The ISO/IEC 9126 does not prescribe which metrics to use. Thus, a concrete distinction between what actually is considered as internal or external quality depends on the selection of metrics and is not necessarily determined by the ISO/IEC 9126. However, this part of the evaluation structure has some similarity to the QADAG even though it is a less concrete representation of quality requirements.

Quality in use refers to quality from a user's point of view. It is organized in a separate hierarchical structure with *effectiveness*, *productivity*, *safety*, and *satisfaction* as topmost characteristics (see Figure 4.10).

In contrast to embedded systems in the automotive industry, the software products focused by the ISO/IEC 9126 may be developed for application by users with different needs. The functionality is not initially fit to an application by a specific user like

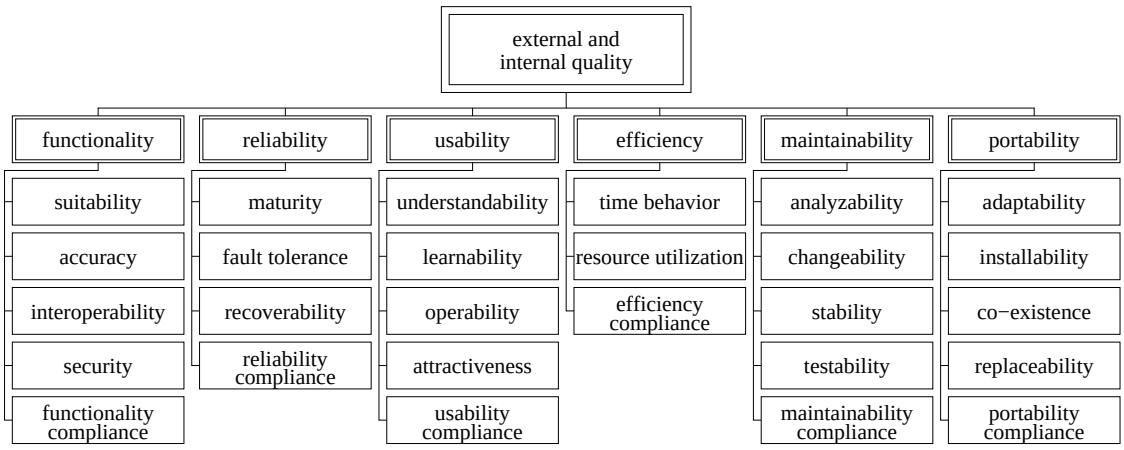

Figure 4.9: ISO/IEC 9126 quality model for external and internal quality

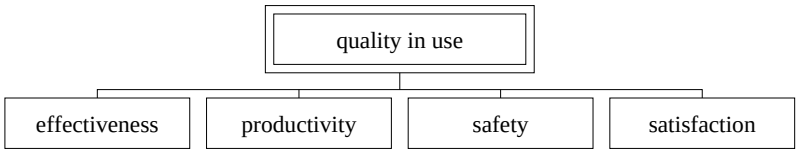

Figure 4.10: ISO/IEC 9126 quality in use model

in the automotive domain. Thus, it depends on the user how well the functionality of the software product fits to the respective needs which are addressed by the characteristics for quality in use. Actually, quality in use can be understood as external quality from the user's point of view although it is separately taken into account by ISO/IEC 9126. The respective characteristics can be integrated in a QADAG instance which then can be considered as user specific QADAG and may need to be changes for different user groups.

Further information on the ISO/IEC 9126 quality model is provided by Part 1 of the ISO/IEC 9126-1 standard [ISO91b]. External metrics for application in the quality model are recommended in ISO/IEC TR 9126-2 [ISO03a]. Internal metrics are suggested in ISO/IEC TR 9126-3 [ISO03b]. Quality in use metrics are given in ISO/IEC TR 9126-4 [ISO04]. A brief overview of the ISO/IEC 9126 is provided by Jung et al. [JKC04].

Architecture Tradeoff Analysis Method - Utility Tree

The Architecture Tradoff Analysis Method (ATAM) has been developed at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA. It is presented in Kazman et al. [KKB+98, KBK+99, KKC00] and Kazman and Klein [KK98].

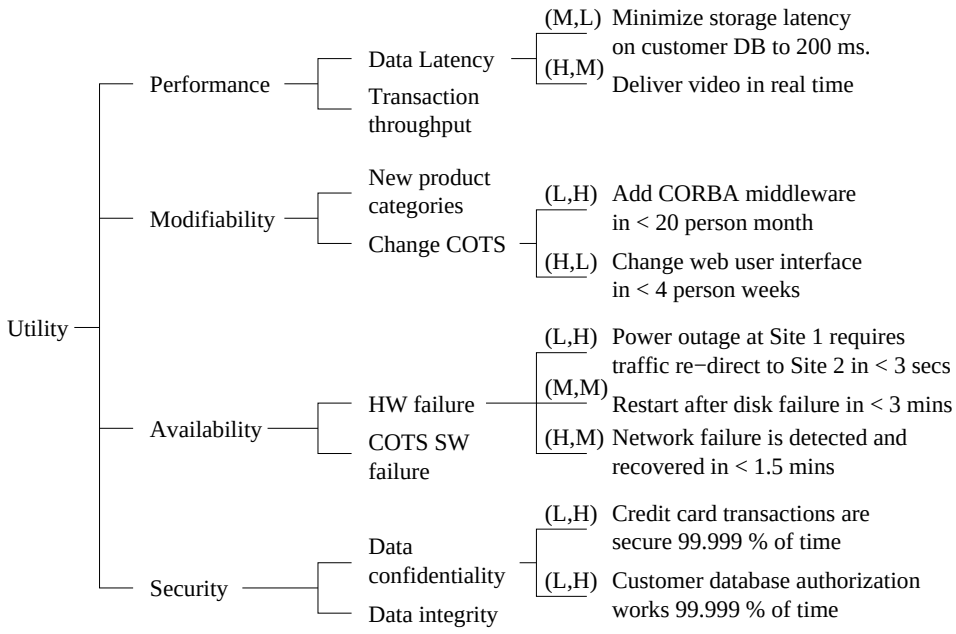

Figure 4.11: Sample utility tree, cf. [KKC00]

It is a scenario-based approach, i.e. scenarios are used to describe the quality requirements to be met by the architecture. The hierarchical structure of quality attributes in ATAM is called Utility Tree. Its leaves are scenarios. This is one of the main differences to the QADAG containing leaf quality attributes as leafs of the hierarchical structure and scenarios as parts of quality attributes. Although the meaning of scenarios is quite the same, ATAM takes scenarios into account without specifying evaluation techniques how to assess the quality regarding the scenarios. The evaluation is usually based on expert knowledge. Scenarios are separated into direct and indirect ones. A direct scenario can directly be executed by an architecture under evaluation. Scenarios, for which an architecture needs to be changed to be executable, are called indirect scenarios (see Kazman et al. [KABC96]). Hence, requirements met by an architecture are represented by direct scenarios. Requirements not yet met by an architecture are represented by indirect scenarios. The consideration of requirements for evaluation by ATAM is concerned with the fulfillment of requirements (scenarios) and not with the hierarchical structuring of quality attributes, which has only an organizational character in ATAM. Priorities of scenarios will become important for tradeoff analysis especially in the context of the Cost Benefit Analysis Method (CBAM) presented in Kazman et al. [KAK01, KAK02]. The application of the Utility Tree in CBAM will be discussed in Section 7.4.

The prioritization of scenarios as step of the ATAM will be addressed in Section 5.3. An example for a Utility Tree as hierarchical structure of quality requirements is

depicted in Figure 4.11 (cf. Kazman et al. [KKC00], Fig. 3). On the right hand side, the scenarios are shown as leaves of the Utility Tree. The bracketed values at the leaves denote the prioritization regarding importance (first part) and the risk regarding the achievement of this scenario (second part). H, M, and L, mean high, medium, and low, respectively.

Goal Question Metric and Factor Criteria Metrics Approach

The Goal Question Metric approach is presented in Basili et al. [Bas93, BCR94]. A *goal* describes a requirement on a product, a process, or a resource like personnel, hardware, and software. Actually, this approach is not only directed to architecture evaluation but provides a structured view on requirements as well. With the goals as topmost quality requirements (cf. the topmost quality attributes in a QADAG), the *question* represents the subattributes which state how to evaluate an e.g. software product. Thus, a question can be considered as leaf quality attribute. Although an evaluation technique in the QADAG usually contains more details than just the metric to be applied, a *metric* in the Goal Question Metric Approach has the same intention as an evaluation technique in the QADAG. A goal is usually based on several questions which are again based on several metrics. A metric can be used by several questions which again may be part of several goals. The hierarchy of this approach is quite flat. However, the idea of a non-tree-like usage of the structure is more strongly intended than it is in the QADAG.

The Factor Criteria Metrics Approach by McCall et al. [MRW83] structures quality requirements in *factors* equal to the topmost quality attributes in the QADAG. A decomposition of factors into *criteria* is available in the QADAG by the decomposition into subattributes. Metrics are used like in the Goal Question Metric Approach. They are covered by evaluation techniques in the QADAG. The structure and decomposition is quite the same as in the Goal Question Metric Approach. The QADAG is compatible to both of them although it supports a considerably more detailed modeling of quality requirements and the application and meaning of metrics regarding the fulfillment of these requirements.

Domain-Specific Software Architecture Comparison Model

An approach addressing the comparison of architectures is the Domain-Specific Software Architecture Comparison Model (DoSAM) by Berger et al. [BRST05]. It is based on a weighted and quantified result for the architecture quality assessed by applying metrics on the architecture variants. However, the quantified results are not only weighted regarding the quality attributes but even regarding the services provided by the architecture (see Figure 4.12). A service can be considered as feature in the automotive domain. With respect to the complexity of automotive systems mostly caused by the amount of features to be realized, an importance-based

Architecture Services	Availability			Modifiability			Weight Service	Sum Service	Total Service
	Weight	Value	Points	Weight	Value	Points			
Data Transfer	20 %	97	19.40	30 %	65	19.50	**24 %**	**81.00**	**19.44**
Data Storage	20 %	57	11.40	30 %	80	24.00	**24 %**	**68.50**	**16.44**
Processing	20 %	65	13.00	5 %	45	2.25	**14 %**	**62.14**	**8.70**
System Management	0 %	10	0.00	5 %	19	0.95	**2 %**	**19.00**	**0.38**
Authent./Author.	20 %	57	11.40	5 %	42	2.10	**14 %**	**54.86**	**7.68**
Presentation	20 %	64	12.80	25 %	90	22.50	**22 %**	**75.82**	**16.68**
Sum QA			**68.00**			**71.30**			
Total QA	**60 %**		**40.80**	**40 %**		**28.52**		**69.32**	**69.32**

Figure 4.12: Evaluation matrix, cf. [BRST05]

weighting of the features is hardly manageable. The DoSAM provides a detailed and service-based view on evaluation results. The complexity of the architecture and the functionality of automotive systems is covered by evaluation techniques of the QADAG that handle the complexity and provide aggregated results regarding a quality attribute. As the QADAG is intended to be a means of communication between stakeholders, its summarizing table view (see Section 6.1.11) is totally sufficient. For a more detailed look inside the architecture evaluation, Architecture Potential Analysis is presented in Section 7.3.

Comparing Quality

In contrast to functional requirements, which are either met or not, extra-functional requirements can be met on different levels of quality. Both, the identification of a value or measurement to be the basis for the quantification and the quantification itself are challenging steps of requirement specification. Moreover, they depend on the application domain and thus cannot be defined in general. As presented in Section 2.1, the ISO/IEC 9126 quality attributes have a strong relation to the effort caused by the application of the software products. The focus is shifted for system qualities identified by Bass et al., for which time is emphasized as central aspect of quality. For embedded systems, the quantification of evaluation results, therefore quality, can be taken into account by interpretation instances provided by experts (see Florentz [Flo06, Flo07b]). These instances map the evaluation result to a quality rate between 0 % (requirement not met) and 100 % (fully satisfied). In the following paragraphs, the comparison of (quantified) qualities will be discussed with respect to the common quality attributes used in different domains (see Section 4.1).

The ISO/IEC 9126 suggests functionality as one quality attribute. This allows for different functionality of different products to be evaluated, which actually leads to a problem for comparison of the software products and their underlying architecture. The question arises, which product to prefer if one system has more functionality

but lacks in terms of how well the functionality is realized and another one realized the functionality quite well but misses to provide some parts of it. Even though a solution for quantification is available, the comparability is strictly limited. Thus, the quality attributes suggested by ISO/IEC 9126 cannot be directly applied in the automotive domain. However, a selection of them may be interesting after some adaption to the specific needs of embedded systems.

For the evaluation of single and individually developed systems as addressed by Bass et al. [BCK98], a comparison of complete system variants is not intended. The evaluation is performed during the development process and only parts of the system are affected by particular development decisions. A quantification may be possible but the interpretation of the results has no point of reference as, i.e. quality differences based on decisions affecting various partial quality results are not comparable either. Kazman et al. [KAK01, KAK02] address this problem in the CBAM approach (Cost Benefit Analysis Method). The approach compares the benefits of different options of investing development effort to improve certain qualities of a system. The CBAM will be discussed in Section 7.4 with respect to the quantification introduced in Florentz [Flo07b].

Architecture variants of an embedded system in the automotive domain are meant to have the same functionality each. Thus, the comparison of several architecture variants is provided. Moreover, the comparison of variants with different functionality can be considered as misleading. With functionality as common basis and quantifiable evaluation results, the achievement of the development objectives, i.e. the fulfillment of the quality requirements, by different architectures can be compared. The explicit representation of the requirements by the QADAG provides comparison of selected requirements. Furthermore, decision support in terms of how to change a variant in order to increase its quality can be provided (see Florentz [Flo07b, Flo07a] and Florentz and Huhn [FH07]). In contrast to CBAM, not the development effort but the development objectives themselves build the center of interest. Thus, tradeoffs directly address the quality results and not the costs for achieving the results like in CBAM. This is not only possible because of functionality as common basis and the quantification of the results but rather because of the development of several architecture variants, which is common for the embedded systems development in the automotive domain.

5 Evaluation Processing Methodology

Architecture development should be guided by the evaluation rather than just being monitored. Consequently, evaluation is more an accompanying activity than a single step. Performing architecture evaluation can be time-consuming and cost-intensive. A reasonable handling of evaluation resources is necessary to get results in time, especially if several architecture variants are to be considered. Based on the expressive representation of the requirements and the explicit structure of the QADAG, evaluation tactics can be defined and decision support can be provided. The methodology presented in this chapter addresses ordered and efficient processing of the evaluation. In case of several architecture variants to be evaluated, the selection of variants to take part in further evaluation and development is one part of the methodology. Actually, this selection is a decision, too. Hence, evaluation methodology and decision support are directly related disciplines in architecture development. Rejecting an architecture variant from the evaluation process is a decision against a combination of certain architectural decisions made earlier which are represented by the architecture variant.

Methodology in the context of this chapter does not describe an evaluation process based on participants, their activities and meetings, documents to be produced, and presentations to be held (cf. Bass et al. [BCK98]). It is rather meant to be a structured procedure of efficiently performing evaluation in terms of saving time and costs. The procedure will be based on the QADAG on the one hand. On the other hand, dimensions of architecture evaluation need to be defined in order to know about the resulting effort and impact of the partial evaluation, i.e. regarding particular qualities. On this basis, tactics can be selected that state when to evaluate which architecture variant with respect to which quality attribute to achieve an overall efficient evaluation process. They are meant to be applied in a heuristic-like manner and have particular objectives like decreasing the extend of the evaluation and handling its complexity. All in all, efficient evaluation means saving costs, time, and personnel resources.

In Section 5.1 dimensions of architecture evaluation are introduced as basis for evaluation tactics definition. In Section 5.2, evaluation tactics are presented. Examples for a possible application in the Body Comfort System case study are provided for each of the tactics. Related evaluation methods are discussed in Section 5.3.

5.1 Dimensions of Architecture Evaluation

The dimensions defined in this section are meant to be a basis for evaluation tactics and decision support. They are directed to evaluation in terms of its structure, its results, and its processing. Because of the highly complex field of architecture evaluation and the immense number of possible considerations, just a selection of dimensions can be taken into account on a certain level of abstraction. Three of them are defined in the subsequent sections (partially introduced in Florentz and Huhn [FH07]). Each of them is divided into subdimensions dealing with the dimension topic in detail. First, the background is presented. Second, important measurements of the subdimensions are discussed. And third, the application of subdimensions in evaluation tactics and decision support is outlined. Examples for the application of the tactics based on the dimensions are given in Section 6.1.

5.1.1 Evaluation Sensitivity

The dimension sensitivity is split up into three subdimensions: the *structural impact* and the *architectural impact* on evaluation results and the *robustness* of evaluation results. The structural impact describes the sensitivity based on the evaluation structure, i.e. the hierarchy of quality attributes. The architectural impact describes the sensitivity regarding architecture artifacts. Figure 5.1 draws a line between composite quality attributes and leaf quality attributes. The first build the hierarchy which leads to the structural impact as it defines the importance of leaf quality attributes by the weightings related to the composite attributes. The latter describe how to evaluate an architecture and thus implicitly determine the architectural impact. To give a concrete statement on evaluation sensitivity, both impacts have to be taken into account in combination. The robustness of evaluation results describes the influence of changing the importance of the quality attributes for a concrete architecture. There is a strong reciprocal relation between impact and robustness. For the latter, an architecture is considered as invariant whereas the evaluation structure changes according to changes of superior development goals. Although the evaluation structure can be considered invariant for a development process, changes of the structure are useful to be considered at least with respect to future development projects.

Structural Impact

The impact of the evaluation structure is based on the hierarchical composition of the QADAG, i.e. the choice of quality attributes with their subattributes and joining techniques. The structural impact can be considered as the absolute weight of a quality attribute if the joining of partial results is proportional. Otherwise, it can be quite difficult to determine the structural impact. For example, let a certain joining technique depend on the actual evaluation results. One subattribute

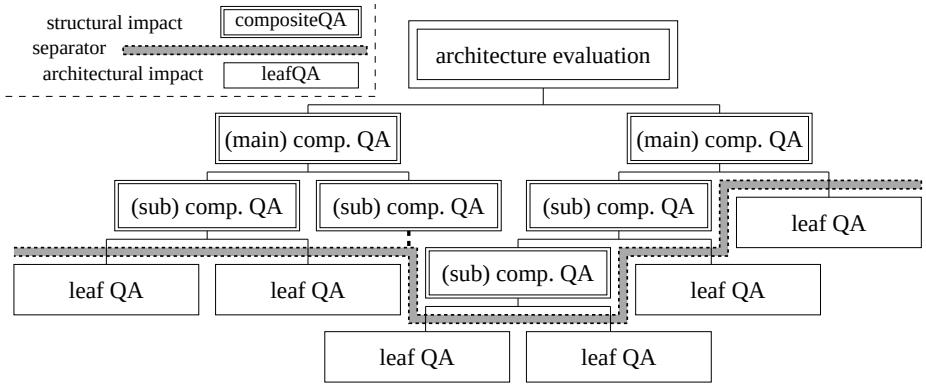

Figure 5.1: Separator between structural and architectural impact in a QADAG

is included quadratically whereas others are included just linearly. In this case, structural impact depends on the architecture as well and is no longer exclusively given by the hierarchical structure of the evaluation. Actually, the structural impact as determined by the evaluation structure can be analyzed in both cases: proportional and non-proportional joining. To simplify matters, the proportional case will be addressed in the following. Non-proportional cases are out of the scope of this thesis. The weighted normalized sum of the subattributes will be calculated which, after all, results in this commonly applied joining technique.

Figure 5.2 presents an example of absolute weights which can be calculated taking the relative importance of a subattribute with respect to all other subattributes of the same composite one into account. Usually, the absolute weight of leaf attributes will be most interesting as architecture variants are evaluated by the techniques located at these attributes.

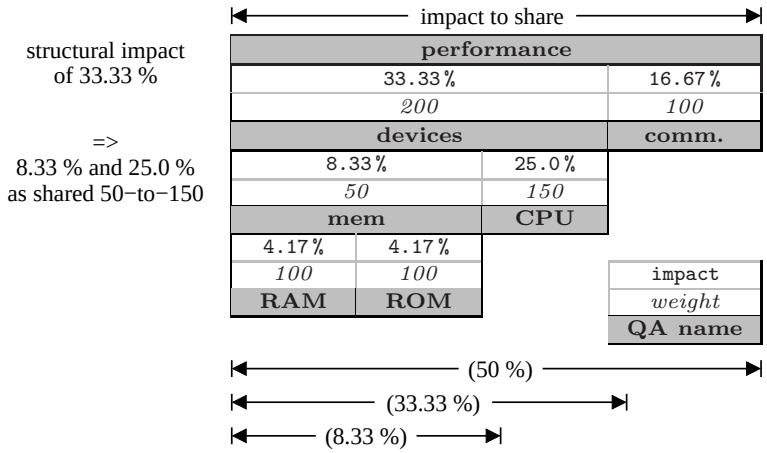

Figure 5.2: Weights and impact in a QADAG

For evaluation tactics, the structural impact is useful for arranging quality attributes according to their relative importance and thus to provide their possible priority for evaluation processing. Great structural impact provides the possibility to have a great share of the overall evaluation result. The more structural impact is covered by a still running evaluation, the more expressive its preliminary result is.

In decision support, the structural impact supports the choice of architecture artifacts to change in order to affect certain quality attributes, namely those with high impact. Again, the architectural impact has to be considered but is individually weighted by the structural impact. Thus, not only an estimation of impact is possible but rather tradeoff analysis based on architectural impact aligned according to results of sensitivity analysis can be performed (see Section 7.3).

Architectural Impact

The architectural impact describes the part of sensitivity that depends on architecture artifacts. The overall evaluation result is hierarchically composed out of partial results. The description of the sensitivities can be quite complicated but not because of the amount of sensitivities or resulting tradeoffs. The most challenging and interesting fact is that sensitivities are not just existent or inexistent, i.e. sensitivity points (see Clements et al. [CKK01]). But rather, dependent on the actual architecture variant, the evaluation result may be influenced in different ways by changing the architecture variant. Hence, the actual sensitivity depends on the architecture variant under consideration.

The description of sensitivity can be quantified in the embedded domain. For example, changing an architecture, which is inured to minor changes, may have less effect than changing one in a more critical area regarding a stakeholder's needs. This means, the sensitivity depends on the actual architecture variant. Figure 5.3 illustrates this issue based on a costs interpretation. Because of the non-linear dependency between costs and quality, the actual value of the architecture variant is a matter of particular interest. The sensitivity, i.e. its architectural impact, strongly depends on the actual architecture variant and may be considerably different for other ones.

For evaluation tactics, the advantage of explicitly investigating sensitivities is the ability to predict evaluation results (see Section 1.3.2). Thus, possible strong insufficiencies of an architecture variant can be detected quite early or even without evaluation therefore avoiding unnecessary evaluation effort.

For decision support, explicit sensitivities are even more valuable than for evaluation methodology. The actual sensitivity of architecture is an important factor in making a decision regarding a possible change of the architecture. Different architecture variants may have different sensitivities as can be seen in Figure 5.3. A differentiated and more detailed look on each architecture variant in a development process allows the correlation of several sensitivities to get quantified statements about trade-

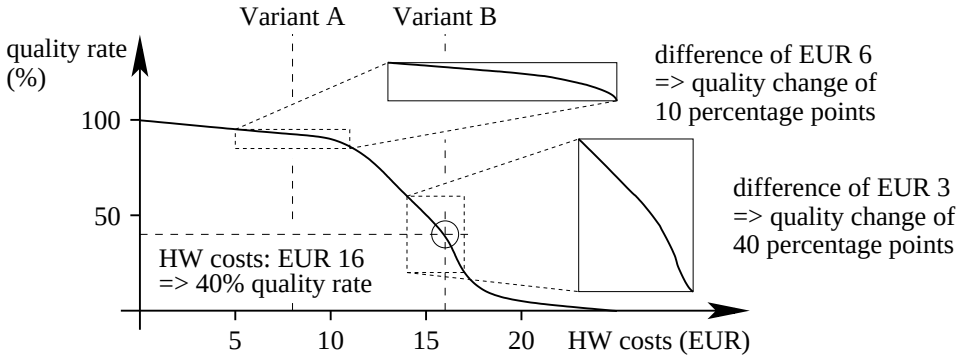

Figure 5.3: Interpretation-based impact regarding costs

offs as needed for architecture analysis (see Chapter 7). Figure 5.4 presents a correlation and thus the tradeoff between costs and performance. A detailed description on how to correlate sensitivities and to use the outcome is given in Section 7.3.

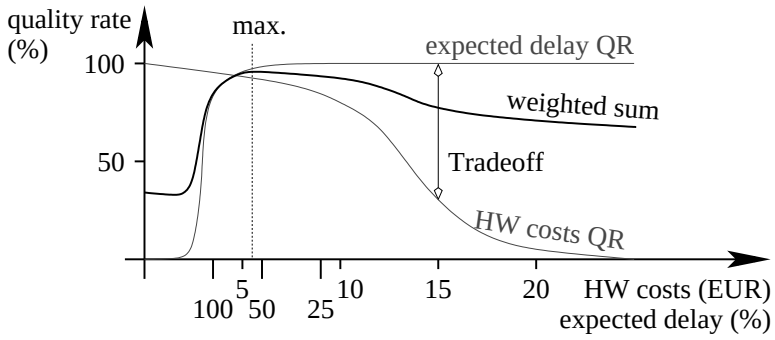

Figure 5.4: Tradeoff between hardware costs and performance

Robustness

To get a proper evaluation result, the structural impact of quality attributes should not change during evaluation. Nevertheless, changes of requirements may occur, for example, as a reaction to altering company goals or shifts of the market. The robustness of an evaluation result describes the effect of such changes on the evaluation result of a concrete architecture variant. The less the quality is affected, the greater the robustness of the evaluation results is. An architecture of high robustness has a greater probability of being reused even after changes of the company goals. Saving development effort is one of the benefits of having a robust architecture especially in domains applying product line approaches (see Etxeberria and Sagardui [ES05] and Bosch [Bos00, Bos06]) for which minor changes are not unusual. Actually, the robustness could be seen as quality of architecture. But, the robustness of the evaluation results is addressed by this subdimension. Thus, the robustness is considered as

dimension regarding the evaluation and not the architecture, i.e. it is no architecture quality attribute.

On the basis of the explicit and quantifiable sensitivities and tradeoffs, the robustness of evaluation results can be determined by considering the partial results affected by the requirement changes. The greater the share of affected partial results, the less the robustness of the results is and vice versa. Actually, the robustness provides no general statements on an architecture variant but is related to concrete requirement changes. The benefit of robustness is clearly located at decision support and less interesting for evaluation tactics.

The robustness of evaluation results should not be confused with the modifiability as a quality attribute. Robustness is strongly directed to evaluation results. Modifiability usually is concerned with changes of the functionality for supporting product lines and future projects. Changes in the context of robustness are changes of the requirements that are not to be changed in order to have a reliable basis for architecture development and evaluation. Nevertheless, changes of the requirements cannot be avoided in practice and thus have to be considered somehow. The intention of robustness is that an architecture should still be suitable even after such changes. Considering robustness as quality attribute is a self-contradiction because high robustness in general would debase the expressiveness of the results achieved with respect to concrete and weighted requirements. A high quality result of an architecture usually has less robustness than a low quality result. After all, high quality means high potential of quality loss if requirements are changed. This conflict clearly exceeds tradeoffs in the evaluation. A robust evaluation result is desirable indeed but has no priority over architecture quality attributes.

The main interest in decision support is related to which architecture variant to choose and which architectural artifacts to change to improve the overall quality. In most cases, architecture evaluation identifies appropriate candidates but is not necessarily definite. Robustness can give the casting hint with respect to future benefits not yet to be accounted in the actual evaluation. For example, two architecture variants reach the same quality rate and thus neither of them can be preferred initially since there would be no advantage in any choice. Scenarios describing changes in the requirements can be used to make a decision. Actually, such scenarios are no architecture quality scenarios as they are concerned with changes of the requirements and not with architecture. Furthermore, considerable differences in the robustness may lead to rethinking of the architecture rationale at least for future projects.

5.1.2 Evaluation Effort

Although architecture evaluation can take place quite early in the overall development process, performing the evaluation should not take too much of the development resources. Even more, the evaluation effort should be kept low in order to provide a quick and efficient decision support in terms of which variants to keep in the

evaluation, which ones to reject, and how to slightly change promising ones in which way if necessary. The evaluation effort has three subdimensions: *extent*, *complexity*, and *input*. Extent and complexity are quite complex ones themselves. Especially for their application in evaluation tactics, a further subdivision is helpful. Actually, the input, which is needed to perform an evaluation, could be accounted to complexity as well. But because of its exposed meaning for building an architecture, it is considered as subdimension on its own. If the input is already used to describe the structural part of the architecture, no additional effort will be caused by using this input for evaluation. If it is exclusively needed by the evaluation, the input will have to be considered in more detail.

Extent

The extent of the evaluation is composed of the *number of architecture variants* to be evaluated and its *scope*. Considering just the number of variants is not sufficient to describe the extent because the size of each variant, i.e. the number of artifacts contained, namely its scope, and artifacts occurring in representations of several architectural decision have a great influence on the overall evaluation effort. Thus, knowing just the number of variants is not that expressive. Furthermore, variants differing in just some details may not cause the same evaluation effort because most of the evaluation results may be alike which may save evaluation effort. Besides the number of variants, the scope is important to denote the extent of an evaluation. The early evaluation in the development process of the embedded domain is meant to select promising architecture variants out of a set. In some cases, new variants are generated on basis of remaining ones or newly-made experiences. The main trend is an early sorting with the effect that evaluation and development will be continued only for selected variants. Thus, the extent changes over the development phase based on the remaining number of variants and their common artifacts. The more decision are represented, i.e. the bigger the scope is, the more different the variants can be, which increases the evaluation effort.

For evaluation tactics, the extent of an evaluation is an important factor. The less extent, the more resources can be spent on more precise evaluation techniques and maybe used for the development of additional variants. Usually, development resources are quite short. Thus, knowledge about extent can help to concentrate the resources to achieve results more efficiently.

Besides additional experiences made by evaluating several architecture variants, evaluation extent has no appreciable influence on decisions support in terms of how to change an architecture variant. Nevertheless, the knowledge of particular architectural decisions and their realization success as variant can be quite insightful. Synergies regarding the architectural artifacts between several variants can consolidate these insights.

Complexity

The complexity is based on the number of evaluation techniques to be performed, which are part of the leaf quality attributes, and the specific effort to evaluate a variant regarding the respective techniques. This specific effort is determined by the *setup* and the *processing*. The setup denotes the complexity in terms of preparing the evaluation by setting up hardware (e.g. test stands, prototypes, etc.) and software to perform the evaluation. The processing describes how complex it is to perform the evaluation. In case of metric-based evaluation, the processing may just be an automated computation which does not need any interaction with experts. In other cases, the processing may be based on expert interaction or even a sequence of meetings keeping several stakeholders busy. Early evaluation may just need a subset of the number of evaluation techniques to select suitable architecture variants. Evaluation taking place late in the development process is usually meant to assure the quality of architecture. Thus, each of the evaluation techniques has to be performed on each of the respective architecture variants.

The complexity may also be increased in case of the setup has to be redone not just for each evaluation technique but for each variant as well. The more complex the setup itself is, the stronger the multiplier effect will be. Hence, setup based on software and models rather than on hardware is to be preferred on early evaluation with many architecture variants participating.

The complexity of certain parts of the evaluation structure can be used to keep evaluation effort low in early evaluation phases. Low complexity, ideally in combination with high impact, allows efficient and expressive estimations about the suitability of an architecture variant. If many K.O. attributes are involved based on low complexity techniques, preferring these attributes might be a reasonable tactic to assure suitability and push the selection of the variants (see Section 5.2).

To estimate the impact of an architectural decision, evaluation of a (partially) changed architecture variant may become necessary. Hence, evaluation techniques with low complexity are preferred over those with high complexity for re-evaluation. Furthermore, techniques with less accuracy and thus less complexity as well may be a possible and helpful substitute in decision support concerns to quickly achieve statements on promising architectural changes.

Input

The necessary architecture models (including their refinements) to provide the information required by the evaluation are called the input of the evaluation. Both, architectural decisions and refinements can be considered as steps in building an architecture variant. Because of architecture being an initial part of the development, the input is not available initially. Although pieces of information may be contained in digital libraries, most of them depend on architectural decisions not yet made.

Besides the evaluation complexity, decision making and information acquisition are constitutive parts in evaluation effort. In particular, input which is additionally needed only by evaluation and not for modeling the architecture should be kept low. Its reuse in the same as well as in further developments is limited. For acquiring and representing further information, additional modeling, measuring, and simulation become necessary which are cost-intensive and need special setup and processing.

The reuse of input directly presented by an architecture variant, i.e. the input of the architecture model, does not cause additional evaluation effort. Especially performance evaluation, which in many cases is based on measuring and simulation, demands performance models and detailed knowledge about the runtime behavior of system components. This kind of input usually cannot be generated and is reusable in few cases only. The evaluation effort is pushed by this kind of evaluation techniques. Thus, they should be performed on variants already succeeded earlier in the evaluation. This decreases the number of variants to be taken into account and avoids waste of evaluation effort on unsuitable ones.

In cases, in which selection is based on evaluation results, the input is not essential for decision support. In other cases, decisions are supported by looking ahead regarding alternatives. Further input may become necessary not just for one but rather various look-aheads. Thus, the less input is needed per look ahead, the more efficient decision support will be performed.

5.1.3 Evaluation Soundness

Soundness addresses the quality of an evaluation result in terms of its validity. To reason about soundness can have different intentions. More soundness means more certainty, i.e. less risk in the selection process of an evaluation. Trying to keep initial evaluation on a number of architecture variants in bounds, soundness may allow earlier rejection of architecture variants seeming to be unsuitable. Especially in the embedded system domain, in which resources are short, not much tolerance regarding their allocation can be granted. Unsound evaluation results may have fatal consequences in terms of resource overload. Thus, the risk of ruling out suitable or even keeping unsuitable architecture variants should be kept low by sound evaluation results. Three subdimensions substantiate soundness as dimension of architecture evaluation: *accuracy*, *susceptibility*, and *challenge* which are addressed in the following.

Accuracy

Especially in the early phases of development, information on the architecture can be coarse-grained or needs to be estimated. Consequently, the evaluation results are coarse-grained and maybe even inaccurate. The accuracy has to be considered because the resources of embedded systems are short. On the one hand, coarse-

grained and estimated information in early phases of development might lead to a potential overload of these resources if there is no adequate amount of tolerance. On the other hand, to much tolerance may lead to wrong and inefficient choices of architectural variants to be taken into account in the ongoing evaluation process. An analogy about how the resources are allocated may help to understand this problem. The resource may be a container which is to be filled with objects. While the size of the objects is known only inaccurately, it is not easy to estimate the amount of objects fitting in the container. In case of small objects, the total number fitting is quite high. Thus, inaccuracies cancel out each other, which lowers the need for additional reserves. In case of bigger objects, the cancel-out effect is decreased drastically or even not existent. The handling of tolerance can be fatal in both ways, providing too much reserves, which leads to too high costs, and providing too less reserves, which leads to technical insufficiencies. Following, the need for accuracy is much higher in the latter case. According to the analogy, lower resource requirements can be e.g. relatively low memory need with respect to the available memory.

As already mentioned, more evaluation accuracy means a less risky selection of architecture variants. Nevertheless, high accuracy may be expensive in terms of time and money which both are to be saved by a well thought-out methodology. Based on the risk accepted in an evaluation process, a tradeoff between accuracy and its costs has to be accepted.

Susceptibility

Susceptibility addresses the possible faultiness of evaluation results as consequence of faulty input. Susceptibility is strongly related to sensitivity. The risk of generating faulty results, i.e. the susceptibility, differs from accuracy as dimension because the resulting inaccuracy is based on faulty input and not on missing detail of the input. Even if information on the architecture is expressed in detail, faulty information will lead to faulty results. Actually, no architect intends to use faulty data. But in case of a highly sensitive evaluation, minor discrepancy has more impact than in case of lowly sensitive ones. This should be taken into account during evaluation for an efficient execution and especially in decision support which is based on sensitivity as well.

The idea of limitation of damage is the most interesting one for an efficient methodology. Architecture variants with highly susceptible evaluation results are more likely to fail in the evaluation process than other ones. Therefore, variants with such evaluation results should be ruled out if a selection is needed anyway and they are not that promising regarding other qualities. Another way to deal with susceptibility can be a closer look to the particular results. There is no general susceptibility of evaluation results because it is based on the sensitivity. Thus, the impact of the faultiness of input depends on the actual architecture variant according to the actual architectural impact (cf. Figure 5.3). Additional selective re-evaluation with further

information lowers the risk of having an unsuitable variant under development. Although architecture is a high-level view with respect to a system implementation, at least in late evaluation, susceptibility effects can be more and more assumed to be quite low because of the maturity of the architecture variants.

Susceptibility as well as decision support are both based on sensitivity, moreover, on the impact of faultiness and changes of the input. Decision support techniques are directly affected by the susceptibility of an evaluation and can be used to investigate it. Decision support has a different intention in early than in late evaluation. In early evaluation, decisions regarding variant selection are most important. In late evaluation, decisions regarding changes of architecture variants build the center of interest. With susceptibility investigation, new possibilities are opened up. Usually, susceptibility effects are higher in early than in late evaluation because of still missing maturity of the architecture. Decision support can be used to investigate susceptibility and avoid misleading selection decisions. With evaluation in terms of quality assurance, the consideration of susceptibility by decision support can be used to denote the risk of architecture changes by missing decision support estimations regarding the expected quality. Following, changes of architecture artifacts causing quite susceptible results may be planed and realized more carefully.

Challenge

The challenge denotes the best achievable quality in an architecture evaluation. The challenge is not necessarily a quality rate of 100 % as competing requirements may be impossible to be completely met in one and the same architecture variant. To determine the challenge of an architecture is quite complex because not all of the tradeoffs caused by competing quality requirements are known. However, knowledge about the challenge is necessary to determine if the valuation of a result is sound. For example, a quality result of 70 % quality rate may be valuated as satisfactory result. Without knowledge on the challenge, this valuation cannot be considered as sound valuation. Maybe results up to 90 % quality rate are possible. In this case, 70 % are not quite good. But if 75 % quality rate already meet the challenge, this result will be very good. The challenge is needed to determine the soundness of the valuation of an evaluation result and not the soundness of the result itself.

A stand-alone and quantified quality result misses expressiveness. Quantified results are expressive only if comparative results of other architectures are available or a reference value or scale, which represents the difficulty to achieve good results, i.e. the challenge, can be given. Tradeoffs are the key to determine the challenge because many strong tradeoffs can cause difficulties to reach a certain level of quality as stated above. The reason for low quality results may be caused by quite competing requirements and not necessarily on bad architecture decisions. This is a very important means of communication between stakeholders. It can be used to explain why some system cannot be realized on an overall satisfactory level. Moreover, cost-intensive

improvement intentions can be put into relation with the potential for improvements with respect to the challenge.

In evaluations with many variants, the challenge is one outcome of the overall evaluation process in terms of experiences on the actual evaluation process. Following, knowledge about the challenge cannot be applied to increase evaluation efficiency. In case of already known challenge, this will help selecting variants to remain in the evaluation process.

The potential of an architecture variant and thus its improvement is in the center of interest of decision support. The challenge allows for concrete statements about the potential of an architecture variant. Therefore, the challenge can be used to stop improvement attempts in time.

5.2 Evaluation Tactics

The time needed for evaluation as well as all other resources like special hardware or even experts are multiplied by the number of architecture variants taken into account in the development process. The evaluation of each of the variants should be performed as efficiently as possible. Actually, the most resource-saving way is to skip an evaluation. Of course, this is intended to be done only if the variant is not promising, which is not known initially. Thus, each of the variants taken into account causes at least some evaluation effort until it can be rejected from the evaluation. Keeping the extent of the evaluation low by reducing the number of variants is possible only if the evaluation is based on the selection out of several variants. In pure software projects, usually just one architecture variant (maybe with some options regarding particular decisions) is available. In this case, a reduction of the number of variants is not applicable. The tactics presented in this section are meant to reduce the number of variants to be evaluated as early as possible in processing the evaluation without rejecting them unseen and avoiding effort-intensive evaluation techniques as long as possible. Thus, the decision whether to reject or continue has to be based on as few partial results as possible to be sure of the rejection and save further evaluation effort. Those results, on which the selection is based, should be gained as efficiently as possible in terms of saving effort and producing accurate results. All in all, the selection decisions are to be made as quickly and as thoroughly as possible.

For evaluation of the case studies, mostly measurement techniques based on metrics will be applied. In the Body Comfort System case study, the evaluation is performed early in the development process. Many variants need to be evaluated with limited availability of input for the evaluation. The intention of this early evaluation is to identify promising variants and reject unappropriate ones in order to continue with just a small set of the variants originally given. The In-Car Radio Navigation System case study, however, is directed to deeper analysis of the evaluation results. The

more detailed modeling of the architecture variants contains executable performance models which provide a calculation based evaluation of architectures. On the same basis, further analysis can be performed to be integrated with the analysis for architecture potential (see Section 7.3). The application of the tactics is exemplified in the Body Comfort System case study.

The tactics can be combined in several ways to achieve a quick reduction of the number of variants. Actually, it strongly depends on the underlying domain and the refinement of the architecture, which tactics and which combination of them are most efficient. However, there is an overall tactic which is not considered in terms of decreasing the extent of the evaluation but more a principle of evaluation processing methodology. According to the hierarchical structure of a QADAG, whose construction is addressed in Section 4.3, a preselection of evaluation techniques to be performed should be kept in mind. The quality attributes located nearby the root of the QADAG represent the superior goals of the architecture, e.g. costs, performance, and so on. Like any composite quality attribute, their quality rate is hierarchically based on the results of their subattributes. In contrast to evaluation techniques, the joining of results is usually less expensive. To get a more precise idea of the quality reached by an architecture variant, evaluation techniques belonging to the same branch of the QADAG, i.e. the same composite quality attribute, should be performed as a group to evaluate the quality of the respective superior quality attributes as whole. Hence, if there is no stronger preference suggested by an applied tactic, this grouping can be very useful. Moreover, evaluation techniques of quality attributes belonging to more than one composite quality attribute are to be preferred as well because they provide input for more than one composite attribute. Note that the QADAG is a directed acyclic graph and not just a tree.

Again, the tactics are meant to save evaluation effort. It might be useful to evaluate variants which are not promising in the overall evaluation but in certain qualities just to gain more experience. Nevertheless, this intention is not addressed by the tactics presented below and should be disregarded until spare evaluation resources are available. The term *architectural tactics* introduced by Bachmann et al. [BBK03] is directed to architecture design and not to evaluation. The objectives of the tactics of this approach are different. Bachmann et al. aim at architectures of higher quality than architectures designed without application of their tactics whereas the tactics represented below are meant to improve the evaluation in terms of efficiency.

In the following, the tactics are presented one by one with a description of how the tactic is meant to work. The description will highlight evaluation dimensions indicating a promising application. Their assets and drawbacks will be discussed as well as a typical application context. Although methodology in terms of applying evaluation tactics cannot be given in general, a heuristic application can be defined with respect to particular needs.

5.2.1 Great Structural Impact

Identifying a set of evaluation techniques holding a certain share of importance to get a sound statement on the most promising architecture variants.

As part of sensitivity, the structural impact (see Section 5.1.1) denotes the absolute importance of a quality attribute, i.e. its weight in most cases, with respect to the overall evaluation. Hence, the more important a quality attribute is, the bigger its contribution to the evaluation result will be. Thus, differences of variants in important quality attributes lead to significant differences in the overall result. A selection of a certain number of variants to be continued in the evaluation should be based on partial results with great structural impact to increase the reliability of the selection.

For evaluations, in which one quality attribute is quite dominant regarding its weight, it may be effectual to evaluate the variants with respect to this one including its subattributes. Usually, this is not the case. Especially in more complex evaluations containing many quality attributes, a good portion of structural impact is spread over several quality attributes. Hence, the structural impact is helpful to identify a set of evaluation techniques to hold a certain share of the total weight/importance. On this basis, a selection of most promising variants can be performed. This tactic is a good example for cases in which to ignore the hierarchical structure in advantage to push the evaluation. Evaluation techniques located at different parts of the QADAG may be preferred over a set within the same branch.

As can be seen in Table 4.5, which is part of the Body Comfort System case study, the communication performance as well as the costs subattributes have high structural impact which qualifies them to be accounted first.

5.2.2 K.O. Attributes

Accounting K.O. attributes first to increase the chance of early rejection of insufficient architecture variants.

Besides structural impact, the architectural impact is interesting to get an idea of an architecture variant's quality without performing a complete evaluation. Knowledge about the sensitivity in terms of architectural impact may not be available in early evaluation. A good substitute is the potential of a quality attribute to reject an architecture variant. Such quality attributes are called K.O. attributes (see Section 4.2). Although sensitivity might not be known in detail at this stage of evaluation, a possible K.O. evaluation of an architecture variant has much impact due to a possible rejection. Additionally, an evaluation regarding K.O. attributes is pretty efficient in most cases because of concrete requirements on which a K.O. is defined. Architecture variants evaluated K.O. will not be suitable. Thus, further evaluation effort can be saved regarding those variants.

Especially, evaluations with a strong constraints satisfaction (see Tsang [Tsa93]) character provide a good basis for this tactic. The selection is based on rejecting architecture variants which will not be buildable regardless their quality in other attributes. Leaving aside possible experiences, useless evaluation effort can be saved to be invested in other architecture variants. The set of potentially effective K.O. attributes may change over architecture variants if various K.O. attributes are contained in the QADAG and problems with particular architecture variants can be identified. Moreover, an architecture variant will be ruled out by its first K.O. result. There is no need to have more than one because it will not be buildable anyway.

K.O. attributes in the Body Comfort System case study are all performance subattributes, the costs, and the battery subattributes. No K.O. attributes are modifiability subattributes and the weight. As a consequence, the first ones are to be preferred by this tactic.

5.2.3 Available Input

Saving effort early in the evaluation to avoid investing too much modeling effort in architecture variants that may be rejected.

Although separately discussed from setup, acquiring information needed as input for the evaluation has a strong setup character (see Section 5.1.2). Information, which is not available as part of an architecture model but has to be provided for evaluation exclusively, significantly increases evaluation effort. Because of the great number of architecture variants in early evaluation, techniques requiring not yet available input may cause gainless effort.

In the case that the input can be reused in later development phases, avoiding evaluation techniques requiring such input means a shift of evaluation effort. With the background of applying additional tactics reducing the number of architecture variants, this shift means an overall decrease of effort because of the savings regarding architecture variants no longer under consideration. The objective of this tactic is not based on quick selection but on deferring effort caused by further evaluation techniques until the selection of architecture variants has been performed.

The weight of the hardware components and the battery of the Body Comfort System are initially known. The subattributes of physics can be evaluated without additional input acquisition because of the available input and the known parameters to calculate the evaluation raw result. Hardware prices will be available as well if no price negotiations are necessary and price lists are on-hand. Usually, CPU, RAM, and ROM information is early available, too. But in the Body Comfort System case study, they depend on the communication mapping and the resulting LIN bus scheduling. Thus, they are not available at first.

5.2.4 Easy Setup

Saving effort early in the evaluation to avoid investing too much in not reusable setup for architecture variants that may be rejected.

The setup of evaluations, mostly with technical background, can be quite complex (see Section 5.1.2). Specific hardware or simulators become necessary and have to be adapted for each of the architecture variants. The effort based on the setup should be avoided if possible because most of it is needed for evaluation only and can hardly be reused in later design phases. Setup for architecture variants, that will not be continued in the development, should be saved by deferring setup-intensive techniques.

For most quality attributes, various evaluation techniques are available providing different accuracy dependent on their input and setup effort (see Section 4.3). Nevertheless, their is a significant discrepancy regarding the applied evaluation techniques in a QADAG. To keep setup effort low, techniques should be ordered regarding their setup effort in case no concurring preference by other tactics has to be considered. First choice evaluation techniques are these with no setup required. The second choice usually is based on COTS (commercial of the shelf) software products. The third one contains techniques including simulators for software and even hardware. The latter techniques require additional hardware models (cf. Available Input tactic above). The most complex techniques are performed on adaptable hardware or even prototypes that have to be rebuilt for each of the architecture variants in the evaluation process. This classification is an example and can be refined for certain purposes.

Neither the Body Comfort System nor the In-Car Radio Navigation System are evaluated by techniques which need special setup. Thus, this tactic is more an outlook for later development phases in which prototypes and simulations are applied.

5.2.5 Easy Processing

Saving effort early in the evaluation to speed up evaluation to quickly get an idea of which architecture variants to be continued.

The processing as part of the evaluation complexity (see Section 5.1.2) has to be seriously considered because it has to be spent for each of the variants. As the Available Input and the Easy Setup tactics, this one has a stronger effort deferring character than a selection intention. Especially for evaluations based on expert and stakeholder meetings, a great number of architecture variants can be very time consuming and even annoying. A previous selection of architecture variants is recommendable.

While processing effort is well known for most evaluation techniques, a respective sorting can be easily done. Again, this tactic relies on a selection of architecture variants by other ones to actually save evaluation effort. Because easy processing,

e.g. by automated evaluation, cannot be provided in many cases, manual evaluation has to be performed. Deferring this until a certain number of architecture variants is rejected from the evaluation process significantly decreases evaluation effort in those cases.

The modifiability subattributes of the Body Comfort System case study are evaluated by experts which requires an expert looking at each of the architecture variants. To avoid this evaluation effort, modifiability should be deferred as long as possible.

5.2.6 Scope Synergy

Preferring evaluations accounting architectural artifacts with multiple occurrences in other architecture variants to achieve synergy effects in evaluation processing.

It is possible that several architecture variants represent common subsets of architectural decisions, which leads to similarity in parts of the architecture, i.e. multiple occurrence of identical artifacts in several architecture variants (see Section 5.1.2). Identifying evaluation techniques concerned with these artifacts enables the evaluation of architectural decisions just once and covers all variants representing them.

This tactic aims at the synergy effect achieved by evaluating equal artifacts of different architecture variants. Decisions represented by the architecture variants build the starting point of this tactic. Thus, the architecture variants have to be taken into account for its application. For some decisions, the evaluation techniques themselves are the key because architectural decisions may not directly aim for specific values of certain architectural artifacts but aim to reach a particular overall quality. Without proper knowledge about those decisions and their influence on the evaluation result, an identification of evaluation techniques to be performed is quite hard. This knowledge has to be provided by the architects that actually have made the decisions. Their intentions contain the particular quality goal represented by a quality attribute that may be equally represented in several architecture variants.

The Body Comfort System case study is based on a legacy system and on three architecture decisions. Although some architecture artifacts are influenced by more than one of the decisions, most decisions are directly reflected in the architecture variants. Thus, the decision variants occur repeatedly in several architecture variants. The communication performance—rarely a candidate to be preferred—is directed to only two LIN schedule variants. The evaluation of one of these variants covers half of the possible architecture variants, which is quite a scope synergy. Weight and the battery subattributes, however, are influenced by several decisions. There are many variants regarding each of these attributes. Hence, these attributes are not to be preferred.

5.2.7 Why Soundness Does Not Contribute

The soundness of evaluation results as addressed in Section 5.1.3 is an important and highly interesting dimension of architecture evaluation. It provides statements on the quality of the results themselves. Nevertheless, it is not considered for defining evaluation tactics for the following reasons:

1. The main intention of the tactics is to save evaluation effort by decreasing the number of architecture variants to be evaluated and by deferring effort until a selection of architecture variants has proceeded. The soundness neither contributes to the first nor to the second kind of tactic.

2. A statement on the soundness is most interesting with respect to evaluation techniques regarding the same architectural artifact and quality requirement. In a QADAG instance, a change of techniques to increase the soundness is considered regarding iterations of evaluation (early and late) in the development progress. During the same iteration, all architecture variants are evaluated by the same QADAG instance providing equal soundness for each of the variants.

3. Actually, a higher soundness is equivalent to the certainty about evaluation results. Although this may be desirable, early evaluation is designed to deal with this uncertainty and so are the tactics, especially selection oriented ones. The selection is never performed careless but as a necessary means of pushing the evaluation. An increased soundness may lead to more certainty but adds few to performing the evaluation more efficiently.

For late evaluation, especially of a single or few architecture variants, soundness becomes more interesting. That is because unsound results may cause fatal mistakes and decisions support and evaluation are growing together. Decisions are no longer used for architecture variant selection but for developing a variant. Thus the soundness of evaluations becomes an important piece of information.

5.3 Related Work – Evaluation Methods

With architecture as essential part of software and system development, architecture evaluation has become important in the development process as well. Several approaches deal with the evaluation of architectures in different domains. Ali Babar et al. [BZJ04], Dobrica and Niemelä [DN02], and Grunske [Gru07] provide an overview of the most common approaches. Ali Babar and Gorton [BG04] and Ionita et al. [IHO02] focus scenario-based approaches as one main domain of architecture evaluation. In the following, the methodology of the current chapter will be compared to one of the most common architecture evaluation and analysis approaches, the Architecture Tradeoff Analysis Method by Kazman et al. [KKB+98, KKC00].

Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) has been introduced by Kazman et al. [KKC00, KKB+98, KK98]. ATAM is a scenario-based approach applying questioning techniques to assess the quality of an architecture. It is directed to a single architecture to be evaluated and not to a set of architecture variants. It describes the whole architecture evaluation process including participants of the evaluation, the outputs of the evaluation, and the steps to be performed. The ATAM can be considered as extension of the Software Architecture Analysis Method (SAAM) presented in Kazman et al. [KBWA94, KABC96] and Clements et al. [CBKA95]. The participants, the outputs, and the steps to be performed will briefly be discussed below.

The participants of an ATAM evaluation process are an external evaluation team, the project decision makers, and the architecture stakeholders. It is quite important to have an external evaluation team instead of an internal one to avoid a lack of objectivity. The evaluation team consists of people with different roles and responsibilities to guide and perform the evaluation. The project decision makers usually are represented by the project manager, an architect of the current project, and maybe a customer representative. The stakeholders of an architecture are representatives of several groups of people interested in the architecture—actually its quality—for certain reasons. Examples for stakeholders are users, operators, developers, customers (buying the software, not actually using it), and so on.

The outputs of the ATAM are a concise representation of the architecture, an articulation of the business goals, quality requirements in terms of a collection of scenarios, a mapping of architectural decisions to quality requirements, a set of identified sensitivity and tradeoff points, a set of risk and nonrisks, and a set of risk themes. Although the architecture to be evaluated is input of the evaluation process, an appropriate representation of the architecture with respect to its intention is one of the outputs of ATAM. This emphasizes the problem of architecture documentation that often is not initially provided in a sufficient way. Moreover, the business goals of the software project need to be articulated again because these goals are not necessarily known by the development team right from the beginning. A set of scenarios will be presented and discussed in the ATAM. They are output of the ATAM in terms of quality requirements. For a proper understanding of architectural decisions, the ATAM provides a mapping between decisions and the quality requirements. The ATAM identifies sensitivity and tradeoff points for the architecture with respect to the fulfillment of the requirements. The sensitivity and tradeoff points will help to guide further development of the architecture. Based on the sensitivity and tradeoff points, risks and nonrisks regarding the architecture and its requirements can be stated. The risks and nonrisks express which requirements may not be met and which ones are considered as safe to be met. A set of risk themes will represent systematic weaknesses of the architecture regarding the identified risks.

The ATAM consists of four phases dealing with preparation, evaluation, continued evaluation, and follow-up. In Phase 0, the evaluation is set up by evaluation team leaders and key decision makers. Phase 1 is the first evaluation phase that usually will take one day to be processed. After Phase 1 and a so-called Hiatus—a break in the evaluation process—of about two to three weeks, the second evaluation phase, Phase 2, starts with a summary of the Phase 1. Phase 2 will take two days to be performed. Phase 3 is meant as follow-up of the evaluation. The evaluation team will produce a final report on the evaluation. Moreover, discussion on what could be done more efficiently in the next evaluation will take place in this phase. The evaluation phases are most relevant in this discussion and are presented in the following. These two phases are performed in six and three steps, respectively.

The six steps of the first evaluation phase are:

Step 1: Present the ATAM
: The steps of the ATAM are briefly presented by the evaluation leader (member of the evaluation team). The process will be explained and the participants are introduced.

Step 2: Present Business Drivers
: The decision makers present the business drivers of the system development, i.e. the functionality, technical, economical, etc., constraints, business goals, the major stakeholders, and the architectural drivers. The latter are represented by the most important quality attributes.

Step 3: Present Architecture
: The architecture is presented by its architects. A presentation of about one hour based on about 20 slides is recommended.

Step 4: Identify Architectural Approaches
: The architectural approaches applied to build the architecture will be identified in this step. The most commonly known type of architectural approaches is the application of specific pattern to achieve certain goals.

Step 5: Generate Quality Attribute Utility Tree
: The root of the Utility Tree is called *utility*. Its children are the quality attributes. The most common attributes are performance, modifiability, security, usability, and availability although this set is not fixed. These quality attributes need to be refined via so-called quality attribute refinements which can be considered as quality attributes themselves. The lowest refinement level—the leaves of the Utility Tree—are the scenarios. The scenarios will be prioritized by the decision makers regarding their importance for the project and the difficulty to be satisfied by the architecture. A scale from 0 to 10 can be used for this prioritizations. However, a scale of high, medium, and low should be

preferred because it is easier to handle and takes less time than the use of the number-based scale. Afterwards, each scenario has a pair of two priorities which are used to generate a prioritized list of scenarios. An example for a Utility Tree is depicted in Figure 4.11.

Step 6: Analyze Architectural Approaches
Relevant architectural decisions are identified as well as sensitivity and tradeoff points and thus risks and nonrisks. On this basis, the architectural approaches identified above will be seriously discussed with respect to their sufficiency and application in the architecture.

After the Hiatus, Steps 7 to 9 are performed as second evaluation phase.

Step 7: Brainstorm and Prioritize Scenarios
Besides the scenarios arranged in the Utility Tree, additional ones may be identified by brainstorming by the stakeholders. These scenarios are prioritized like the ones already contained in the Utility Tree.

Step 8: Analyze Architectural Approaches
According to Step 6, the newly discovered scenarios are mapped to the architectural decisions relevant for them by the architects.

Step 9: Present Results
The results to be presented in this step are (see Bass et al. [BCK98]):

1. Architectural approaches documented
2. Set of scenarios and their prioritization from the brainstorming
3. Utility Tree
4. Risks discovered
5. Nonrisks documented
6. Sensitivity points and the tradeoff points found

The ATAM is directed to the process of quality assessment of an architecture. Although many (until its application) uncovered requirements will be identified and analyzed, an explicit modeling of the requirements as provided by the QADAG is not the intention of the ATAM. The ATAM has a more process-oriented view on architecture evaluation. The QADAG is used to support the evaluation process but does not define the whole process. However, the underlying ideas of a structured evaluation and representation of requirements and how to meet them are the same.

The evaluation methodology presented for the QADAG has a different idea of evaluation processing than the ATAM. The ATAM is concerned with one architecture and which parts to improve or change to meet a set of requirements. One main

benefit of the ATAM is that the participants of the ATAM evaluation become aware of the requirements, not yet considered problems and risks, and how to achieve the architecture and business goals. The Utility Tree is mainly used as auxiliary structure to generate scenario priorities. According to the priorities, the available development resources are spent to meet the requirements represented by the highest prioritized scenarios. Thus, the efficiency intention is directed to a reduction of the development effort taking only to the most important requirements into account. The methodology of evaluation processing presented with respect to the QADAG is meant to deal with a set of architecture variants to be efficiently evaluated. The requirements are explicitly given and usually not discussed anymore during evaluation. The main intention is how to identify the most promising architecture variants (or a small set of them) in a short period of time without causing to much expenses. A consideration of only the most important requirements is not intended like in the ATAM. The whole set of requirements represented by quality attributes will be kept in mind at least for the most promising architecture variants. All in all, Phases 1 and 2 of the ATAM try to keep further development effort low which is considered to be quite short (cf. Bass et al. [BCK98]). The evaluation effort is not in the center of interest of these phases. However, Phase 3 of the ATAM is meant to reduce the evaluation effort of future ATAM processings by increasing the efficiency of the interaction of the evaluation participants. However, this is not based on the evaluation structure as backbone like in the tactics presented in Section 5.2, which are meant to increase the efficiency of the evaluation processing.

6 Evaluation Case Studies

Two case studies have been introduced in Chapter 3. The Body Comfort System case study is an example for a selection-based architecture evaluation. The In-Car Radio Navigation System case study shows the capabilities of the approach for late evaluation taking more detailed architecture models into account.

According to the selection-based character of the first case study, the application of the evaluation tactics presented in Section 5.2 is motivated with respect to the quality attributes applied in the Body Comfort System QADAG. This QADAG has been constructed in Section 4.3 and will now be used in the case study evaluation.

The In-Car Radio Navigation System case study with its three variants will be evaluated applying the Modular Performance Analysis (MPA) introduced by Wandeler et al. [WTVL06]. The results of that approach will even be reused for analysis of the overall evaluation results as will be shown in Section 8.2. The case study QADAG will be introduced in Section 6.2. The table view on the weights and impact of the quality attributes are presented in Table 8.4, Section 8.2.

In Section 6.1, the Body Comfort System case study is evaluated. The evaluation results are presented at the end of that section and in Appendix B. The In-Car Radio Navigation System case study is evaluated in Section 6.2.

6.1 Body Comfort System Evaluation

The Body Comfort System (BCS) case study is concerned with architecture variations based on extensions of a legacy system. It has been developed in cooperation with the VOLKSWAGEN AG (cf. Mielke [Mie07]) and has been introduced in Section 3.1. The main intention of this evaluation is to identify the most promising variant out of the available ones. Because of the great number of architecture variants—and this case study covers only a relatively small part of a complete automotive embedded system—the evaluation process is aimed at selection of the variants. To achieve a time-saving and cost-efficient evaluation, the methodology presented in Chapter 5 is taken into account. Although a complete evaluation of each of the variants is performed, the tactics to be applied in order to push the evaluation are pointed out with respect to the evaluation techniques. In the subsequent sections, the evaluation of the variants according to the case study QADAG in Figure 6.1 is presented.

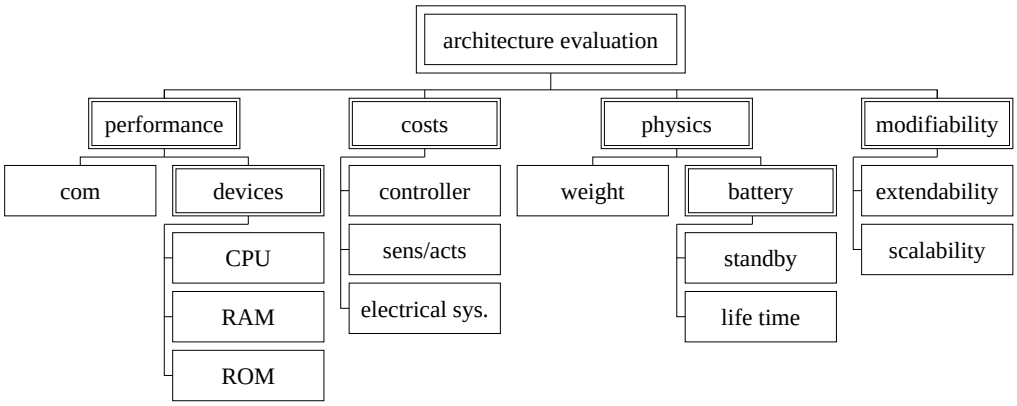

Figure 6.1: BCS case study QADAG

6.1.1 Communication Performance

The performance of the communication of the Body Comfort System variants is evaluated by calculating the bus utilization, i.e. average bus load. Liu and Layland [LL73] introduced a method to calculate the utilization of communication resources by

$$U = \sum_{i=1}^{m} (C_i/T_i) \tag{6.1}$$

with U as the utilization, C_i as the time needed for the transmission of signal i, and T_i as the transmission period of signal i. This calculation has to be performed for both buses in the system, i.e. the LIN bus and the CAN bus. The worse result will be taken as the evaluation technique's result because that result may become a bottle-neck causing too big delays. According to the function mapping, the signals to be communicated via the buses are taken into account. Their width in bits and an estimation of their periodic appearance in combination with the transmission rate of the bus (e.g. LIN with 19.2 bit/s) are input for the calculation (see Tables A.1 and A.3). The signal appearance is assumed to be periodic. Hence, no scenarios are taken into account as evaluation driver. Nevertheless, the utilization does not consider any overhead like message headers etc. And although the messages will be sent periodically, message collisions and inefficient utilization may lead to jitters. To take these facts into account, the bus utilization will be multiplied by a factor for the message overhead and one for inefficient utilization as can be seen below.

A comparison of the evaluation results of the LIN and the CAN bus of this case study has shown that the LIN bus has a higher utilization which leads to worse results. They will be taken into account as conservative result of the evaluation technique for communication performance.

The LIN specification [LIN06] allows to adjust the rate for the messages to be transmitted. Two different communication mappings with respect to the LIN are

contained in the case study. For architecture variants in which the Short Lift Control is mapped to the Power Window Devices, a different communication mapping (see Table A.8) is needed than for the architecture variants with Short Lift Control on the Body Control Device (see Table A.9). To meet the high requirements of the Short Lift Control—quick response time to release the window from its grooves before the door is opened—additional messages with short periods of 40 ms have to be inserted into the scheduling. The short periods let the utilization of the bus raise from 14.32 % (SLC on BCD) to 14.56 % (SLC on PWD). Moreover, the integration of additional (short period) messages leads to additional message headers but to a better utilization of the user data space in the messages. The utilization of the LIN bus with 10 ms period by user data of 14.32 % is multiplied by 1.94 to take the message headers into account and by 1.8 to take the suboptimal utilization of the user data space of the messages into account. The 5 ms scheduling of the second LIN schedule for the variants with Short Lift Control on Power Window Devices defines more (and shorter) messages. This leads to additional message headers taken into account by a factor of 4.0. The utilization of the user data space is quite efficient and represented by a factor of only 1.1. The results of the evaluation are $14.32\,\% \cdot 1.94 \cdot 1.8 \approx 50\,\%$ and $14.56\,\% \cdot 4.0 \cdot 1.1 \approx 65\,\%$. The interpretations of the bus utilization according to the interpretation instance depicted in Figure 6.2 are 98 % and 34 %, respectively. A utilization of more then 80 % will lead to a K.O.

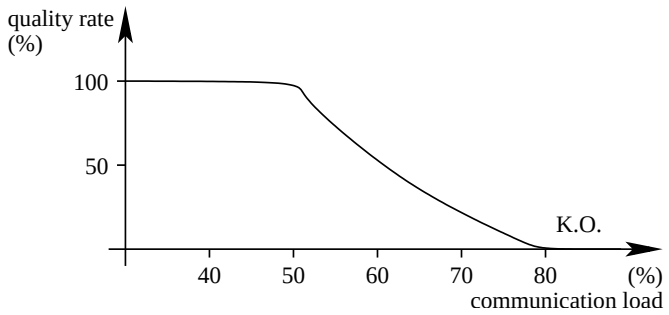

Figure 6.2: BCS communication utilization interpretation

Only two variants of the communication need to be taken into account by the communication performance evaluation. This is the best premise for the application of the Scope Synergy tactic (see Section 5.2.6). All 18 architecture variants are covered by only two variants regarding the LIN schedule (see Tables A.8 and A.9) based on the communication mapping. Although the evaluation regarding the communication performance might have higher effort than others, the scope synergy will help to keep the overall effort low. Moreover, the great structural impact of this quality attribute may motivate the application according to the Great Structural Impact tactic (see Section 5.2.1).

6.1.2 CPU Performance

The CPU performance is based on utilization as well. The main difference to communication performance is the static utilization which is assumed for RAM and ROM utilization, too. Thus, the resource is not dynamically allocated but statically, i.e. a function with a certain need for some resource will allocate this amount permanently. Conflicts between different functions sharing the same resource do not need to be taken into account. Nevertheless, the evaluation is done in the architecture development, when accurate resource requirements of the implemented function are still unknown. Nearly complete utilization should only be aspired if accurate resource requirements are available and only a segregated part of the system (say subsystem) is affected for which sufficient resources are granted.

As hardware resources can hardly be shared across controllers, each of the controllers have to be separately evaluated. The worst result will be taken as overall result of the evaluation technique. Controllers, for which the resource requirements are accurately known and which can be considered segregated parts of the system, may not be taken into account for the overall result as long as they are still acceptable and although they may be worse than others.

The input needed for evaluating the CPU performance are the resources available on the controllers (see Table A.5), the resources required by the function (see Tables A.3 and A.2), and the function mapping to determine which function needs resources on which controller. The ratio between available and required resources can easily be calculated. In this case study, the Power Window Devices can be considered segregated as well as the Convertible Top Devices (if available). Their CPU performance will not be taken into account for aggregation of the overall result of this evaluation technique as long as they do not cause any problems, i.e. lead to a K.O. of the architecture variant. Because of the static allocation of the resources, no scenarios are needed to drive the evaluation. However, software like the operation system etc. will decrease the available resources, which in fact has to be taken into account although such software is no functions in the literal sense.

Table 6.1 contains the input for the evaluation of the CPU performance of the Body Control Device as well as the results. There are only four architecture variants (instead of 18) regarding this controller as can be seen in the table. The mapping of the Short Lift Control to the Body Control Device requires to shorten the LIN periods from 10 ms to 5 ms. The LIN master software needs more processor power at 5 ms, which leads to a bigger difference than just the 1 % of the Short Lift Control.

As already stated above, the CPU performance of the Body Control Device will be taken as representative because the other controllers can be considered segregated and will not lead to a K.O. of architecture variants. The interpretation of the results according to the interpretation instance of Figure 6.3 leads to 100 %, 100 %, 98 %, and 95 % quality rate, respectively. A utilization of more than 100 % will lead to a K.O.

type	CPU util	ratio	mapped to BCD			
PCC	2.80 MIPS	35 %	×	×	×	×
SLC	0.08 MIPS	1 %		×		×
CTC	0.40 MIPS	5 %			×	×
CLC	0.16 MIPS	2 %	×	×	×	×
misc	0.88 MIPS	11 %	×	×	×	×
LIN master 10 ms	0.16 MIPS	2 %	×		×	
LIN master 5 ms	0.32 MIPS	4 %		×		×
RF receive	0.16 MIPS	2 %	×	×	×	×
OS 16 bit	0.64 MIPS	8 %	×	×	×	×
BCD variants	sums					
standard	4.80 MIPS	60 %	⇐			
with SLC	5.08 MIPS	63 %		⇐		
with CTC	5.20 MIPS	65 %			⇐	
with SLC + CTC	5.44 MIPS	68 %				⇐

Table 6.1: CPU performance of the Body Control Device (8 MIPS)

The CPU performance attribute is promising for the Available Input tactic (see Section 5.2.3) as the input needed usually is known from earlier projects or can be taken from component libraries in future projects. Nevertheless, the communication mapping needs to be available before starting the evaluation because this may have some effect on the CPU utilization as well. Thus, the evaluation cannot be brought forward that much with respect to the application of evaluation tactics.

In this case study, the CPU utilization is indirectly influenced by the communication mapping because the LIN master is parametrized according to the mapping requirements. In case a 5 ms period is needed, the LIN master causes a higher CPU utilization of 0.32 MIPS instead of only 0.16 MIPS as for the 10 ms period (see Table A.2).

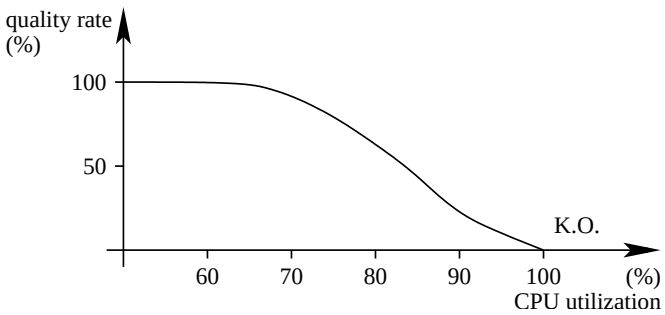

Figure 6.3: BCS CPU utilization interpretation

6.1.3 RAM Performance

As for CPU performance, a static utilization is assumed for RAM performance. Dynamic allocation is not yet implemented for embedded automotive systems because of the relatively high risk compared to the small benefit. Most functions do not need much memory exclusively for calculation which could be freed afterwards. States of the internal behavior of functions and representations of the actuators states (like e.g. power window position) have to be held in memory permanently anyway. Access time is not yet considered in this evaluation but may be integrated in later evaluation.

Memory cannot be shared across controllers. Thus, the controllers have to be evaluated in separate. Again, the worst result will be taken as overall result for the evaluation technique. Controllers, which can be considered segregated (see above), can be omitted in the aggregation as long as they do not lead to a K.O. of the variant under consideration.

The input needed for this evaluation contains the RAM capacity of the controllers (see Table A.5), the RAM required by the functions (see Tables A.3 and A.2), and the mapping of the functions. Again, because of the static allocation, no scenarios are needed to drive the evaluation. The results will be rounded up to avoid pretending accuracy that, in fact, is not available at the architecture development state.

type	RAM util	mapped to BCD			
PCC	320 byte	×	×	×	×
SLC	30 byte		×		×
CTC	200 byte			×	×
CLC	120 byte	×	×	×	×
misc	2420 byte	×	×	×	×
LIN master 10 ms	100 byte	×		×	
LIN master 5 ms	120 byte		×		×
RF receive	150 byte	×	×	×	×
OS 16 bit	180 byte	×	×	×	×
BCD variants	results				
standard	68 %	⇐			
with SLC	69 %		⇐		
with CTC	70 %			⇐	
with SLC + CTC	71 %				⇐

Table 6.2: RAM utilization of the Body Control Device (8 kbyte)

The evaluation results regarding the Body Control Device RAM are listed in Table 6.2. Again, only four variants of the Body Control Device have to be taken into account due to the mapping of the Short Lift Control and the Convertible Top Control.

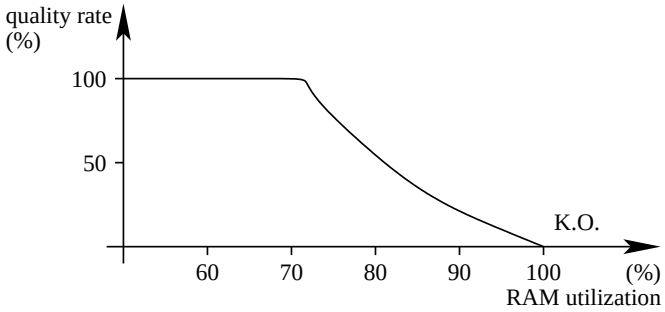

Figure 6.4: BCS RAM utilization interpretation

The results of 68 %, 69 %, 70 %, and 71 % utilization are interpreted to 100 % quality rate as can be seen in Figure 6.4. A utilization of more than 100 % will lead to a K.O.

The RAM performance attribute is promising for the Available Input tactic (see Section 5.2.3) as the input needed usually is known from earlier projects or can be taken from component libraries in future projects. Like for the CPU performance attribute, the communication mapping needs to be done before starting the evaluation because this may have some effect on the RAM utilization as well.

In this case study, the RAM utilization is indirectly influenced by the communication mapping because the LIN master is parametrized according to the mapping requirements. In case a 5 ms period is needed, the LIN master causes a higher RAM utilization of 120 byte instead of only 100 byte as for the 10 ms period (see Table A.2).

6.1.4 ROM Performance

The evaluation of RAM and ROM (and CPU) are quite alike in this early state of architecture development. One main difference is that the static allocation of ROM will hardly be substituted by dynamic allocation in future development.

Again, the worst result of the controller specific evaluation is taken as overall result for the evaluation technique. And again, controllers, which can be considered segregated (see above), can be omitted in the aggregation as long as they do not lead to a K.O. of the architecture variant under evaluation.

The ROM capacity of the controllers (see Table A.5), the ROM required by the functions (see Tables A.3 and A.2), and the mapping of the functions build the input needed for ROM performance evaluation. Again, because of the statical allocation, no scenarios are needed to drive the evaluation. In contrast to CPU and RAM evaluation, software inactive during standard runtime mode needs to be considered (represented by the flash software for system updates in this case study). They do allocate some of the ROM even when inactive. The results of the evaluation will

be rounded up to avoid pretending accuracy that, in fact, is not available at the architecture development state.

type	ROM util	mapped to BCD			
PCC	35840 byte	×	×	×	×
SLC	3072 byte		×		×
CTC	12000 byte			×	×
CLC	5000 byte	×	×	×	×
misc	68520 byte	×	×	×	×
LIN master 10 ms	2500 byte	×		×	
LIN master 5 ms	3000 byte		×		×
RF receive	9500 byte	×	×	×	×
OS 16 bit	15000 byte	×	×	×	×
BCD variants	results				
standard	73 %	⇐			
with SLC	74 %		⇐		
with CTC	77 %			⇐	
with SLC + CTC	79 %				⇐

Table 6.3: ROM utilization of the Body Control Device (256 kbyte)

The evaluation results of the ROM are contained in Table 6.3. They are interpreted to 95 %, 91 %, 75 %, and 64 % quality rate for a ROM utilization of 73 %, 74 %, 77 %, and 79 %. A utilization of more than 100 % will lead to a K.O. as can be seen in Figure 6.5.

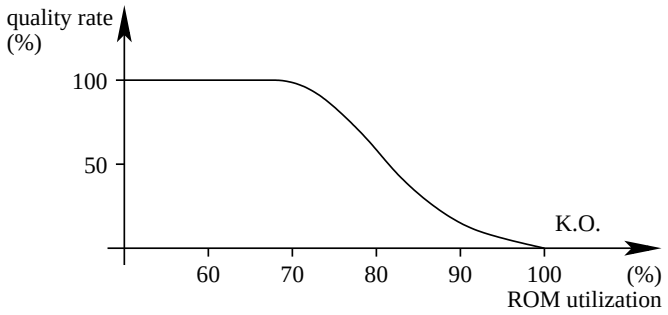

Figure 6.5: BCS ROM utilization interpretation

As for the CPU and RAM performance attributes, the ROM performance attribute is promising for the Available Input tactic (see Section 5.2.3) as the input needed usually is known from earlier projects or can be taken from component libraries in future projects. Again, the communication mapping may influence the utilization,

which impedes bringing the evaluation forward in the evaluation process with respect to the application of evaluation tactics.

In this case study, the ROM utilization is indirectly influenced by the communication mapping because the LIN master is parametrized according the mapping requirements. In case a 5 ms period is needed, the LIN master causes a higher ROM utilization of 3000 byte instead of only 2500 byte as for the 10 ms period (see Table A.2).

6.1.5 Costs

The costs of a system are the total costs of all its hardware components and the battery. Moreover, installation costs can be taken into account which are not covered neither by the architecture model nor by the evaluation.

The evaluation of costs significantly differs from other quality attribute evaluations of this case study. Although the costs quality attribute is a composite one, the actual interpretation of its subattributes results takes place after they have been joined. This is not just possible but rather recommended because costs are a type of resource that can easily be shared by several components. A counter-example are hardware resources like memory. Free memory resources cannot be used by another controller which may be short of memory according to the functions mapped on it. However, expenses saved by some component may be spent on another one. Hence, the costs for the hardware are taken into account in combination. They are weighted equally as none of them is more or less important than the others.

The input needed to sum up the overall costs of the system are the costs of the components used to build one system. While sensors, actuators, controllers, and the battery have a cost property assigned (see Tables A.4, A.5, and A.7), the costs for the cable harness need to be calculated on basis of its weight (see Table A.6) multiplied with the current market value of copper. Especially in market segments of high volume production, the hardware costs for a system are strictly limited in order to achieve a cost-efficient production and a profitable product. The costs to be spend are given as interpretation instance as discussed below.

Although taken into account in combination, the costs are separately calculated as defined by the case study QADAG (on page 106) to keep track of the costs and put effects of architecture changes into relation. Resulting changes of the costs for controllers of e.g. €1 indeed are to be considered in a different way than changes of the same size of sensors and actuators. Usually, the costs for them are about the fifth part of the controller costs in this case study.

Figure 6.6 depicts the interpretation of costs. The coordinate system does not contain the whole interpretation but the interesting range. The quality rate axis is intersected at nearly 80 %. That does not mean that there is no possibility to reach 100 % at all but the architecture variants of this case study are out of that range. Actually, the cost pressure of high volume manufacturers is reflected in high

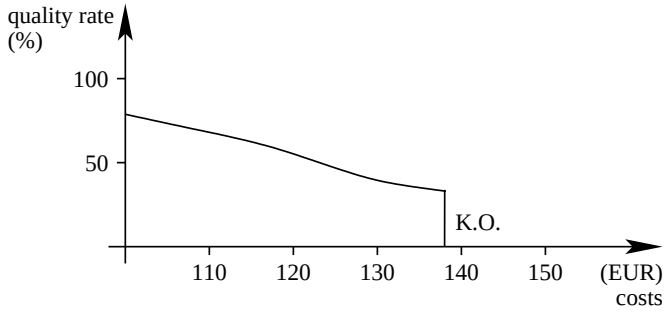

Figure 6.6: BCS costs interpretation

requirements and thus in demanding interpretations as depicted above. Assuming that prices will decrease, an adjustment of the costs interpretation can be taken for granted as well. The costs quality attribute is a K.O. attribute. The 0 % interpretation of €138 and above will reject the respective variant. In this case study, 6 out of 18 variants (those with Reduces Convertible Top Device) are ruled out because of too high costs. Even the least expensive variant with costs of €111 reaches a quality rate of just 67 %, which emphasizes the cost pressure in this application domain.

The costs quality attribute is a favorite of the most evaluation tactics. It has great structural impact, the input is available and it is processed without too much effort. Moreover, it is a K.O. attribute which may reject inadequate variants and thus becomes a candidate for the K.O. Attributes tactic (see Section 5.2.2).

This case study does not take software costs into account because they are not considered in the original development process either. First, software and especially its costs are not visible in the still applied controller-oriented process because these costs are contained in those of the controllers. Second—and in the function-oriented process—, the software components can be taken from a digital library and can relatively easily be installed on different hardware platforms (cf. AUTOSAR, see Heinecke et al. [HSF+04]). In this case, not additional software unit costs need to be taken into account (ignoring an apportion of development costs). A detailed discussion of the consideration of costs in the evaluation is content of Section 7.4.

6.1.6 Weight

The weight of the system is measured by adding the weight of its electrical system consisting of hardware components and the battery. The weight of a single component alone is not expressive because it is the overall weight which is crucial.

The weight of the hardware components (see Tables A.5 and A.4) and of the battery (see Table A.7) are the input for this evaluation. The weight of the cable harness is based on the length and the cross-sectional area of the communication lines multiplied with the density of copper (see Table A.6). The limit of the system

weight is implicitly given by the interpretation. In contrast to e.g. RAM as resource, there is no concrete resource limiting the weight of the system. This evaluation technique emphasizes the importance of evaluation documentation. Otherwise, there is no chance to reproduce the quality result of this evaluation because the resource limit is not part of the architecture model.

As the weight is evaluated with respect to the entire architecture and numerous components are needed to build it, architecture variants with equal evaluation result are quite rare. However, the application of the evaluation is very simple. Hence, the evaluation is not presented with all the results in detail here (they can be seen in the results in Section 6.1.11) but are discussed by means of selected architecture variants. The biggest battery (72 Ah capacity) is the heaviest as well. Variants with this battery have a weight between 22.3 kg and 22.8 kg. The medium battery (61 Ah capacity) variants have a weight between 19.3 kg and 19.8 kg. The architecture variants with the smallest battery (50 Ah capacity) have a weight between 18.3 kg and 18.8 kg.

Although the main difference of the results is caused by the battery, the admittedly small amount of weight of other components needs to be taken into account as well. First, most of the components like communication lines are quite numerous. Second, their overall share of weight is an important means to put their actual weight into relation in order to identify potential to save weight. There is no reason to neglect this potential only because some heavy components are part of the architecture. Nevertheless, the heavy ones may be the first choice in order to decrease weight. After all, they have the most potential for saving weight.

The interpretation in Figure 6.7 results in the following quality rate ranges. The 72 Ah variants are in the range of 73 % and 79 %. The 61 Ah variants are in the range of 93 % and 95 %. The 50 Ah variants are in the range of 96 % and 98 %.

The weight quality attribute is a candidate for the the Available Input tactic as the weight of the components is initially known and rarely changes over time. Additionally, it is easy to process and does not require any setup before its application.

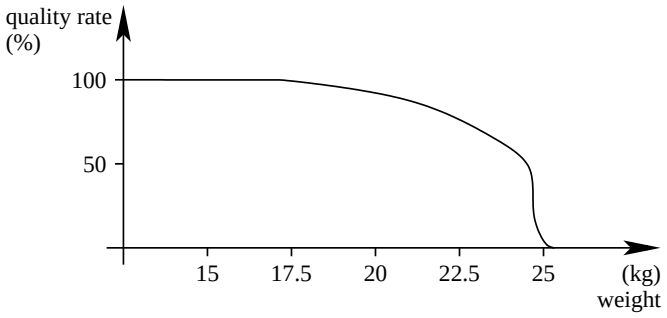

Figure 6.7: BCS weight interpretation

6.1.7 Battery Standby Time

The standby time of the battery is calculated by Equation 6.2

$$t_{battstandby} = \frac{q_{battery} \cdot 0.35}{i_{standby} \cdot \frac{24\,h}{d}} \tag{6.2}$$

with $q_{battery}$ as the capacity of the battery in ampere-hours and $i_{standby}$ as the standby current of the system in ampere. The standby current of the system is the sum of the standby currents of the system's devices (see Table A.5).

The battery standby time is directed to a permanent standby, e.g. until it will be impossible to start the engine with the remaining battery power. The battery needs to be charged about 40 % in order to provide enough power. Too low charge leads to a power drop. The battery is considered fully charged between 80 % and 100 % of its nominal capacity. 80 % is the lower limit of charge before a recharge is initiated. With the 40 % remaining charge to start the engine in mind, the factor in Equation 6.2 represents 80 % - 40 % - 5 % (energy reserve) = 35 %. This factor refers to the part of the battery capacity that is available for standby in the worst case based on the lowest limit for full charge and energy reserve taken into account.

devices	standby current	variants (SLC and CTC mapping)					
		BCD BCD	BCD Mini.	BCD Red.	PWD BCD	PWD Mini.	PWD Red.
PWD	0.25 mA	4	4	4	2	2	2
PWD (ext.)	0.35 mA	0	0	0	2	2	2
BCD	4.50 mA	1	1	1	1	1	1
MiniCTD	0.12 mA	0	1	0	0	1	0
RedCTD	0.02 mA	0	0	1	0	0	1
surveillance	6.00 mA	1	1	1	1	1	1
others	6.56 mA	1	1	1	1	1	1
standby current in mA		18.06	18.08	18.18	18.26	18.28	18.38
battery variants		battery standby time (rounded down)					
50 Ah		40 d	40 d	40 d	39 d	39 d	39 d
61 Ah		49 d	49 d	48 d	48 d	48 d	48 d
72 Ah		58 d	58 d	57 d	57 d	57 d	57 d

Table 6.4: Battery standby time of the Body Comfort System

Table 6.4 contains the standby current of the devices and their occurrence in the variants for Short Lift Control and Convertible Top Control. On this basis, the standby current for the variants can be calculated which will be the input for the

battery standby time with respect to the battery variants as shown in the bottom part of the table. The results are rounded down because only complete days are taken into account.

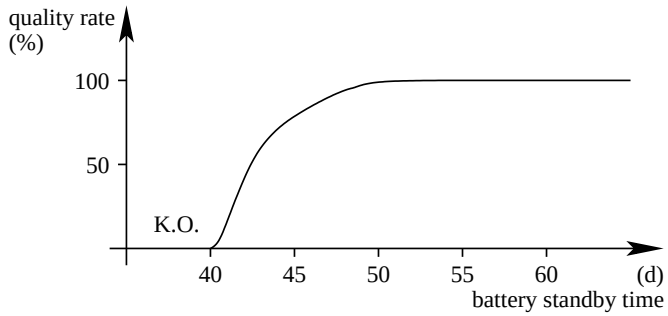

Figure 6.8: BCS battery standby time interpretation

The interpretation instance shown in Figure 6.8 starts with 1 % quality rate for a minimum of 40 days standby time. As the battery standby time quality attribute is a K.O. one with the minimum requirement of 40 days standby, every architecture variant with a standby of 39 days or less will result in a K.O. In this case study, the architecture variants with Short Lift Control on the extended version of the Power Window Device and a 50 Ah capacity battery do not reach the required 40 days and therefore are rejected.

The battery standby time can be evaluated quite easily in terms of input acquisition and processing. Thus, it is preferred by the Available Input tactic (see Section 5.2.3) and the Easy Processing tactic (see Section 5.2.5).

6.1.8 Battery Life Time

The battery life time depends on the battery capacity in ampere-hours and the daily current in ampere per day. The daily current is calculated by Equation 6.3 as the standby current for $t_{standby} = 22.7\,h$ (a day) and a
follow-up energy of $f_{followup} = 3.875\,Ah$ (a day).

$$i_{daily} = \frac{t_{standby}}{d} \cdot i_{standby} + \frac{q_{followup}}{d} \tag{6.3}$$

Equation 6.4 calculates the life time of the battery in years.

$$t_{battlife} = \frac{q_{battery} \cdot 100}{i_{daily} \cdot \frac{365\,d}{y}} \tag{6.4}$$

The battery capacity is multiplied by 100 to get the overall capacity provided by the battery in its life time of 100 full charging cycles. The daily current is multiplied by 365 days per year to get the yearly current. The result is the life time in years.

variants	standby current	daily stby current	daily current	battery life time		
SLC, CTC				50 Ah	61 Ah	72 Ah
BCD, BCD	18.06 mA	0.410 Ah	4.285 Ah	3.19 a	3.90 a	4.60 a
BCD, Mini.	18.08 mA	0.410 Ah	4.285 Ah	3.19 a	3.89 a	4.60 a
BCD, Red.	18.18 mA	0.413 Ah	4.288 Ah	3.19 a	3.89 a	4.60 a
PWD, BCD	18.26 mA	0.415 Ah	4.290 Ah	3.19 a	3.89 a	4.59 a
PWD, Mini.	18.28 mA	0.415 Ah	4.290 Ah	3.19 a	3.89 a	4.59 a
PWD, Red.	18.38 mA	0.417 Ah	4.292 Ah	3.19 a	3.89 a	4.59 a
daily standby time: 22.7 h daily follow-up current: 3.875 Ah						

Table 6.5: Battery life time of the Body Comfort System

The results of applying the equations mentioned above are shown in Table 6.5. Although there are differences in the daily current, they have no significant effect on the results. The battery capacity, however, has major influence on the results, i.e. about 250 days between the battery variants.

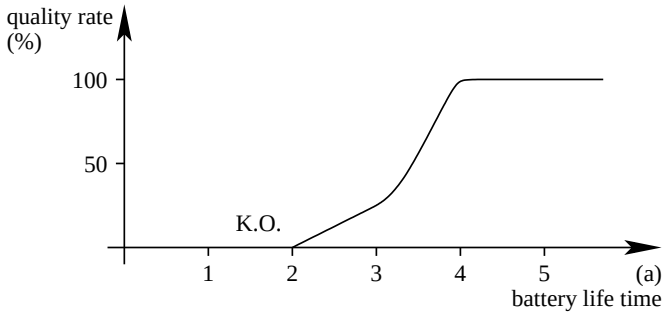

Figure 6.9: BCS battery life time interpretation

Figure 6.9 contains the interpretation of the battery life time which is oriented at the requirement of at least 2 years (implied warranty) and at best at 4 years or more. A life time less than 2 years leads to a K.O.

The battery life time can be evaluated quite easily in terms of input acquisition and processing as well. The input acquisitions contains the modeling of user behavior to estimate the frequency and duration of the systems runtime. Nevertheless, it is candidate for the Available Input tactic (see Section 5.2.3) and the Easy Processing tactic (see Section 5.2.5).

6.1.9 Extendability

The evaluation of the extendability is based on expert knowledge. The architecture variants are shown to an expert who, actually based on facts and metrics, forms an opinion. The focus of this evaluation technique cannot be stated initially because it depends on the expert's opinion which parts of the architecture are essential for the evaluation and which are not. In this case study, the expert's focus lies on free resources of the controllers mainly based on memory (ROM) load. Hence, there will be subresults for different controllers which need to be aggregated. The Body Control Device and the Power Window Devices in the front are taken into account. While extendability is speculative, it is quite hard to give a meaningful aggregation for the subresults. After all, the extension to come are not yet known. However, it is unlikely that the front Power Window Devices will be affected differently. Thus, the extendability will be based on both types of controllers in equal shares although there are double as many front Power Window Devices than Body Control Devices.

The resource capacities of the controllers in combination with the resource needs with respect to the function mapping are the input needed for this evaluation. Actually, the resource needs of the functions have to be derived from the needs of the functions mapped to the respective controller. As this evaluation technique is based on expert knowledge, the result will be a quality rate.

The evaluation of the Body Control Device results in 100 % extendability if neither the Short Lift Control nor the Convertible Top Control is mapped to it. Thus, resources for further extensions are still available. If one of the functions is mapped to the Body Control Device, 50 % extendability will be determined by the expert. Both functions mapped to the Body Control Device lead to 0 % extendability. The evaluation of the Power Window Devices seems to be a paradox at the first sight. Mapping the Short Lift Control to the front Power Window Devices leads to more extendability (100 %) than leaving the devices untouched (0 %). But the change of the devices is the key. The Power Window Device without Short Lift Control has sufficient resources for its actual configuration but has no resources left for extensions. Mapping the Short Lift Control to the device requires additional memory which is available only in specific amounts. Hence, the extension of the memory (ROM) exceeds the need for additional memory, what leads to unused resources for further extensions. As already mentioned, the evaluation result of this technique is aggregated to equal shares of the subresults. A Convertible Top Device is not taken into account because it is applied only for special purpose and thus rarely considered for extensions.

Because the extendability result is based on expert knowledge, the result is already given as quality rate. An interpretation instance is not necessary. Moreover, no raw value to be interpreted is available because the expert knowledge provides no such output.

6.1.10 Scalability

Like extendability, scalability is evaluated based on expert knowledge. Scalability is directed to up- and downsizing the functionality of the system without adding new features. Existing features may be extended. Examples are additional power windows in other bodies or different grades of automation of powered tops. They have been developed starting with manually controlled soft tops via powered soft tops to powered hard tops and even ones including a sliding roof today. Moreover, the Body Comfort System may be deployed in other bodies without convertible top. In these cases, the Convertible Top Control becomes obsolete. In another scenario, the Convertible Top Control may be scaled up to support powered tops. Additional control for the actuators driving the top will become necessary depending on its type.

However, such control is quite complex and the actuators require a certain amount of power to move the heavy top. Besides computation resources, power has to be provided with respect to the Convertible Top Control. Thus, a location of the control close to the actuators is desired to avoid long power providing communication lines in case a powered top needs to be supported. The architecture variants with the Convertible Top Control mapped to the Body Control Device neither meet the first nor the second requirement of scalability. These variants are evaluated to 0 % scalability. The architecture variants with a Convertible Top Device avoid long power providing communication lines because of the device positioning at the rear end of the body. Furthermore, the Reduced Convertible Top Device is at least capable to control simple powered tops like e.g. soft tops. Thus, those variants are evaluated to 80 % scalability whereas the variants containing a Mini Convertible Top Device are evaluated to 70 % scalability because of the lack of computing resources. None of the variants reached a quality of 100 % in scalability because the Convertible Top Devices of the more scalable variants are designed only for various soft tops. Nowadays, convertible hard tops become available requiring special actuator control which is not covered by the devices considered in this case study.

Again, no interpretation is needed as the scalability evaluation technique is based on expert knowledge directly providing quality rates as result.

6.1.11 Evaluation Results

The results of the Body Comfort System evaluation are listed in Table 6.6 (ordered by rank). The K.O. of some of the variants is represented by crossed out results. Evaluation result tables of four architecture variants are exemplarily presented in this section. The four architecture variants are (1) the favorite of the evaluation (see Table 6.7), (2) the best one without additional Convertible Top Device (see Table 6.8), (3) the best one for which only the Body Control Device was changed and no additional Convertible Top Device is needed (see Table 6.7), and (4) the one with least quality but no K.O. (see Table 6.7). Actually, the latter is also the least

expensive one which makes it a top favorite for the management. The results have been presented in parts with respect to their quality attributes. The overall results are presented and analyzed in detail in Chapter 8.

rank	result	variant			on page
		SLC	CTC	battery	
1	70.6 %	PWD	MiniCTD	61 Ah	121
2	69.6 %	PWD	MiniCTD	72 Ah	189
3	67.3 %	PWD	BCD	61 Ah	122
4	65.3 %	PWD	BCD	72 Ah	189
5	61.5 %	BCD	MiniCTD	61 Ah	190
6	60.1 %	BCD	BCD	61 Ah	122
7	59.5 %	BCD	MiniCTD	72 Ah	190
8	58.0 %	BCD	BCM	72 Ah	190
9	54.6 %	BCD	MiniCTD	50 Ah	191
10	52.5 %	BCD	BCD	50 Ah	122
11	63.4 %	PWD	MiniCTD	50 Ah	191
12	60.2 %	PWD	BCD	50 Ah	191
13	56.0 %	PWD	redCTC	72 Ah	192
14	55.9 %	PWD	redCTC	61 Ah	192
15	47.6 %	PWD	redCTC	50 Ah	192
16	42.3 %	BCD	redCTC	72 Ah	193
17	42.2 %	BCD	redCTC	61 Ah	193
18	34.0 %	BCD	redCTC	50 Ah	193

Table 6.6: BCS evaluation results overview

Variant SLC on PWD, CTC on MiniCTD, 61 Ah										
70.6 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
98.2 %			38 %			89.2 %			85.0 %	
-			€132			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	98.3 %		-	-	-	94 %	86.0 %		100 %	70 %
50 %	-		€92	€17	€23	19.6 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	95 %				92 %	80 %		
	60 %	68 %	73 %				48 d	3.89 a		

(weights)
QA name
quality rate
result

Table 6.7: Evaluation results Variant SLC on PWD, CTC on MiniCTD, 61 Ah

Variant SLC on PWD, CTC on BCD, 61 Ah										
67.3 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
94.5 %			48 %			89.6 %			37.5 %	
-			€124			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	91.0 %		-	-	-	95 %	86.0 %		75 %	0 %
50 %	-		€84	€17	€23	19.4 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	98 %	100 %	75 %				92 %	80 %		
	65 %	70 %	77 %				48 d	3.89 a		

(weights)
QA name
quality rate
result

Table 6.8: Evaluation results Variant SLC on PWD, CTC on BCD, 61 Ah

Variant SLC on BCD, CTC on BCD, 61 Ah										
60.1 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
60.2 %			64 %			92.3 %			0.0 %	
-			€113			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
34 %	86.3 %		-	-	-	95 %	90.5 %		0 %	0 %
65 %	-		€78	€13	€22	19.3 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	95 %	100 %	64 %				96 %	85 %		
	68 %	71 %	79 %				49 d	3.90 a		

(weights)
QA name
quality rate
result

Table 6.9: Evaluation results Variant SLC on BCD, CTC on BCD, 61 Ah

Variant SLC on BCD, CTC on BCD, 50 Ah										
52.5 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
60.2 %			67 %			48.2 %			0.0 %	
-			€111			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
34 %	86.3 %		-	-	-	98 %	15.0 %		0 %	0 %
65 %	-		€78	€13	€20	18.3 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	95 %	100 %	64 %				1 %	29 %		
	68 %	71 %	79 %				40 d	3.19 a		

(weights)
QA name
quality rate
result

Table 6.10: Evaluation results Variant SLC on BCD, CTC on BCD, 50 Ah

6.2 In-Car Radio Navigation System Evaluation

The In-Car Radio Navigation System (ICRNS) case study is based on three architecture variants (see Figure 3.9) built from scratch and is mainly directed to performance evaluation. Thus, not just particular performance requirements are to be expressed but rather performance analysis of the variants becomes possible. The results of the analysis will be used as partial result in the evaluation and as input for the analysis of architecture potential as will be shown in Section 7.3. Besides performance, cost-efficiency and modifiability are also objectives of the system development. Hence, they are topmost quality attributes of the case study QADAG presented in Figure 6.10.

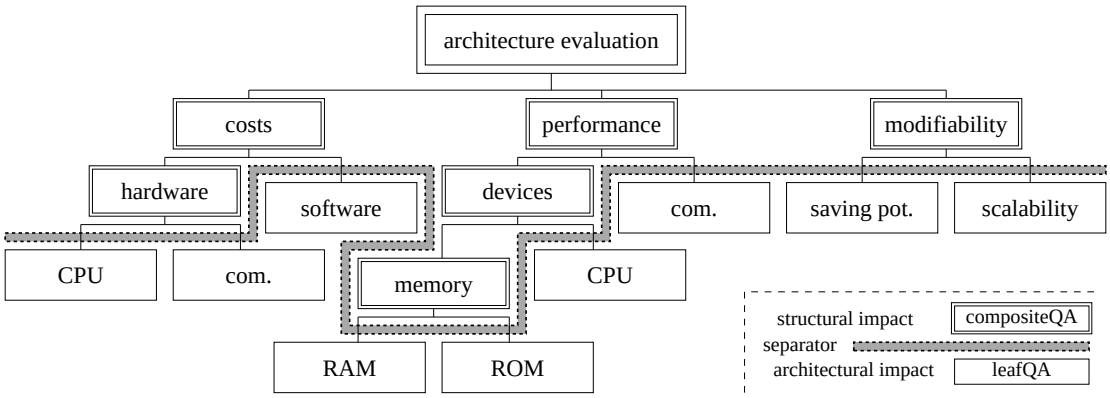

Figure 6.10: ICRNS case study QADAG

To keep track of the overall evaluation results as well as for further analysis of the variants, the weights and the impact of the quality attributes are given by Table 8.4. However, the weights are contained in the evaluation results tables at the end of this section as well. The partial results, which are joined according to the weights, are assessed by evaluation techniques attached to the leaf quality attributes of the QADAG. The joining is based on weighted and normalized summation. The evaluation of the architecture variants with respect to the quality attributes in combination with their required input are presented in the subsequent sections.

Figure 3.9 contains Variants I, II, and III of the In-Car Radio Navigation System introduced and analyzed by Wandeler et al. [WTVL06]. Besides the underlying architecture, the Modular Performance Analysis (MPA) of Wandeler et al. will be adopted in this evaluation. The MPA will shortly be introduced in the context of architecture analysis in Section 7.2.

6.2.1 Costs

As can be seen in Figure 3.9, the architecture variants differ in the underlying hardware architecture. The implementation of the functions is assumed to be invariant

regarding the controllers on which they are mapped. Thus, software costs are considered without details. Variants I and II are based on a controller network which leads to additional costs in contrast to Variant III which is based on a single controller. Furthermore, the computation power will serve as basis for controller costs. Actually, the prices are contrived which does not effect the presentation of the possibilities of this approach. Moreover, prices will change over time anyway. Thus, the documentation of them is an important means to understand back-dated architectural decisions.
The variant prices are:

Variant I: *€16 (€10 + €1 + €5 (network))*
Variant II: *€14 (€4 + €5 + €5 (network))*
Variant III: *€10*

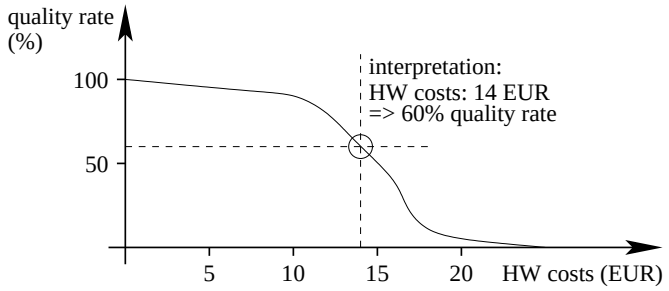

Figure 6.11: ICRNS costs interpretation

The interpretation of hardware costs is depicted in Figure 6.11. Variant III with costs of €10 is the most affordable one, the one with costs of €16 is least affordable. Thus, €10 will be interpreted to 90 % quality rate. Actually, €16 is still affordable as the interpretation will be 40 % quality rate. €14 is interpreted to a quality rate of 60 %.

6.2.2 Performance

Although the performance of computation hardware (devices) and of communication hardware are related, they can be separately taken into account. Thus, a more detailed analysis can be applied on the architecture based on more specific evaluation results. The simple scheduling algorithm of Liu and Layland [LL73] (s.a.) can be used to assess bus utilization. Again, RAM and ROM are mostly statically used in the automotive domain. Their evaluation can be done on basis of the scheduling algorithm or by simple utilization calculation taking resource capacity and requirements of the functions into account. The CPU usage is quite a bit more difficult to evaluate. Especially in early design phases, only few details of resource requirements

and scheduling are available. In the Body Comfort System case study, a legacy system has been reused, which leads to the advantage of available experience data. If such data is not available, experts will have to estimate the CPU utilization. With growing amount of detailed information, approaches like the MPA can be applied in the evaluation process and used to calculate the CPU utilization, which leads to increased accuracy of the results.

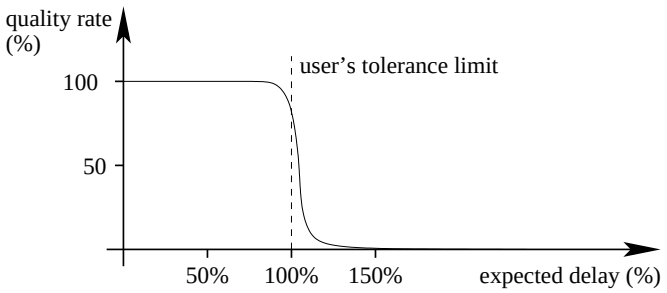

Figure 6.12: ICRNS expected delay interpretation

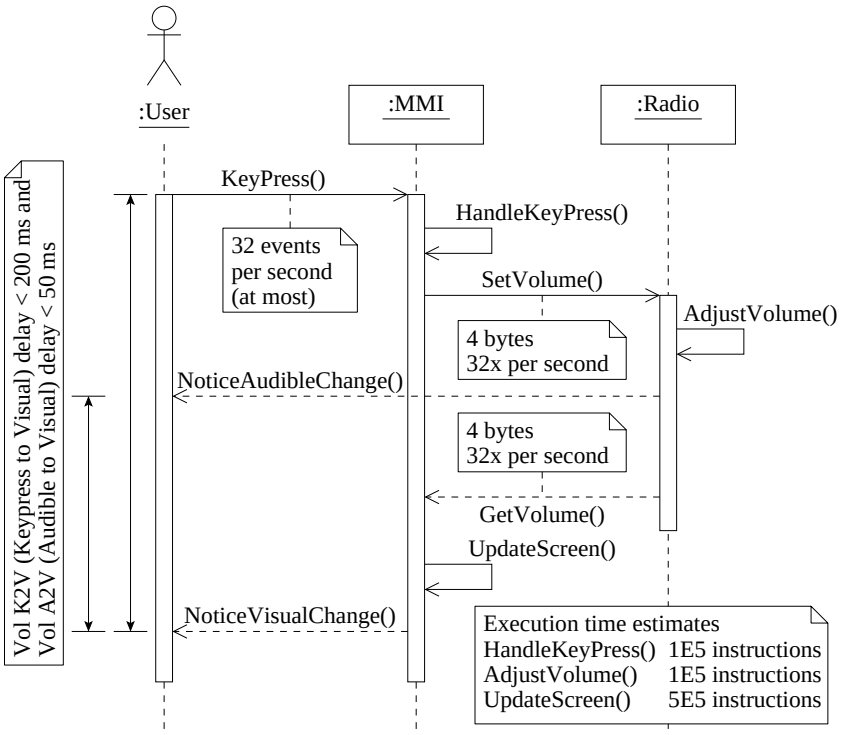

Figure 6.13: Change volume scenario, cf. [WTVL06]

MPA is based on the behavior of the system directed to the time needed for execution and communication. While multimedia mass data is not considered in the evaluated architectures (there are additional communication lines for such data), the delay of communication and the following computations are most interesting. Delay in the communication will lead to delay of the start of computation. Furthermore, the computation itself will take time, too. The Traffic Message Channel (TMC) may cause computation-intensive reactions of the system, i.e. recalculation of the route. The user will tolerate some delay for such calculations. But in case of a change of the radio volume, the reaction should be immediately noticeable to save the user's patience. Thus, the system performance can be measured by the reaction delay. An interpretation of the expected delay is given in Figure 6.12. The expected delay is given as percentage of the user's tolerance to cover various expectations regarding different use cases. MPA works on use case scenarios (for communication behavior) represented by sequence diagrams containing timing requirements (see left hand side of Figure 6.13). In combination with detailed information on time and computation power consumption of the messages and computations as well as knowledge of the hardware resources, an MPA model (performance model) can be created for each architecture variant. This model supports analyzing (evaluating) the performance of the respective variant.

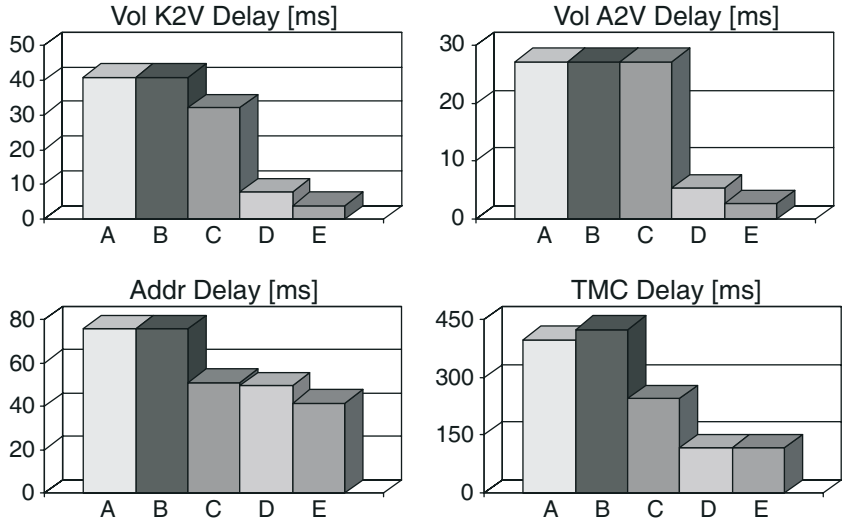

Figure 6.14: Maximal end to end delays, see [WTVL06]

MPA results state that the requirements are met by all architectures. Thus, a 100 % quality rate is assigned for each variant for CPU performance as well as for communication performance as can bee seen in Figure 6.14 taken from Wandeler et al. [WTVL06]. It contains the MPA results with respect to different scenarios for which the MPA has been performed. In the figure, Variants C, D, and E are

Variants I, II, and III, respectively. The benefits of integrating the MPA are presented in Section 8.2 in which the results are considered in detail.

6.2.3 Modifiability

Sufficient performance for all variants in combination with consideration of the costs quality attribute qualifies the least expensive variant to be realized. This is no surprise and represents most business strategies. The evaluation of the architecture modifiability shows the benefits of taking several quality attributes into account. A growth scenario attached to the modifiability quality attribute defines that the architecture should be reusable in product lines targeted at lower budget. Thus, the navigation system can be left out of the architecture while radio and MMI (man-machine interface) are still necessary. Let Variant II and III be favorites because of their low costs. If the navigation system is left out, the hardware architecture of Variant II will be scaled down to a single controller architecture. Variant III already contains only one controller. The performance will not be affected negatively because the computation-intensive navigation system is no longer part of the system (scalability: `ok`). Variant II can get rid of the unused capacities and thus of costs (saving potential: €9, i.e. 100 % quality rate), Variant III can not (saving potential: €0, i.e. 0 % quality rate). Variant II without navigation system is only €5 which is half the price of Variant III. 100 % quality rate will be the modifiability result for Variant II. 50 % quality rate will be the modifiability result for Variant III because a modification is possible even if not cost-efficient. Variant I has the same problem like Variant III and reaches 50 % quality rate for modifiability, too.

6.2.4 Evaluation Results

Besides specific diagrams representing e.g. technical evaluation results, a summation of the results is represented with respect to the QADAG in Tables 6.11, 6.12, and 6.13.

They are the basis for discussion regarding the importance of quality attributes, i.e. their weight. As seen above, the most cost-efficient Variant III is ruled out by the more expensive Variant II. The weights reflect the requirement for at least partial reuse of the architecture in other products or product lines with the background of saving expenses for future system development. Omitting the modifiability, Variant III will be the most promising one.

The results are input for further analysis introduced in Section 7.3 and applied in Section 8.2. In the latter, the discussion of which architecture variant may be the favorite after changing some details will arise again. Up to now, decision support was directed to selection of variants. The analysis results will lift it to decision support in terms of how to change architectures to push the evaluation results, i.e. identify unused architecture potential.

Variant I											
80 %											
-											
50			*100*					*50*			
costs			**performance**					**modifiability**			
70 %			100 %					50 %			
-			-					-			
100		*100*	*200*			*100*		*100*	*100*		
HW costs		**SW costs**	**devices**			**com**		**sav.pot.**	**scalab.**		
40 %		100 %	100 %			100 %		0 %	100 %		
€16		-	-			MPA:ok		€0	ok		
100	*100*		*50*		*150*						
devices	**com**		**mem**		**CPU**						
-	-		100 %		100 %						
€11	€5		-		MPA:ok						
			100	*100*							
			RAM	**ROM**							
			100 %	100 %							
			-	-							

(weights)
QA name
quality rate
result

Table 6.11: Evaluation results Variant I

Variant II											
95 %											
-											
50			*100*					*50*			
costs			**performance**					**modifiability**			
80 %			100 %					100 %			
-			-					-			
100		*100*	*200*			*100*		*100*	*100*		
HW costs		**SW costs**	**devices**			**com**		**sav.pot.**	**scalab.**		
60 %		100 %	100 %			100 %		100 %	100 %		
€14		-	-			MPA: ok		€9	ok		
100	*100*		*50*		*150*						
devices	**com**		**mem**		**CPU**						
-	-		100 %		100 %						
€9	€5		-		MPA: ok						
			100	*100*							
			RAM	**ROM**							
			100 %	100 %							
			-	-							

(weights)
QA name
quality rate
result

Table 6.12: Evaluation results Variant II

Variant III											
86.25 %											
-											
50			*100*					*50*			
costs			**performance**					**modifiability**			
95 %			100 %					50 %			
-			-					-			
100		*100*	*200*				*100*	*100*	*100*		
HW costs		**SW costs**	**devices**				**com**	**sav.pot.**	**scalab.**		
90 %		100 %	100 %				100 %	0 %	100 %		
€10		-	-				MPA: ok	€0	ok		
100	*100*		*50*		*150*						
devices	**com**		**mem**		**CPU**						
-	-		100 %		100 %						
€10	€0		-		MPA: ok						
			100	*100*					*(weights)*		
			RAM	**ROM**					**QA name**		
			100 %	100 %					quality rate		
			-	-					result		

Table 6.13: Evaluation results Variant III

7 Architecture Analysis

In this section, architecture analysis is presented in terms of investigating the sensitivity of architecture quality. The knowledge of sensitivities will be used to explain architecture evaluation results and improve the development process by supporting architectural decisions. One main intention is to identify parts of an architecture which will seriously affect the evaluation result if changed. Undesired effects and risks can be avoided by well thought-out changes. Decisions regarding changes of an architecture can be guided to achieve better evaluation results. Potential of an architecture variant can be uncovered. The architecture knowledge gained by architecture analysis can be used in the current as well as in future projects. It will help to understand an architecture and to document and communicate the rationale of an architecture. Furthermore, development effort can be saved as architecture components and even whole parts of an architecture can be reused, decisions can be supported, and mistakes can be avoided.

In Section 7.1, the term architecture sensitivity is introduced and brought into relation to terms like dependency and tradeoff. The Modular Performance Analysis and its results with respect to further architecture analysis are presented in Section 7.2. How to identify relevant dependencies as starting point for Architecture Potential Analysis is introduced in Section 7.3. Related approaches are discussed in Section 7.4.

7.1 Architecture Sensitivity

As architecture elements and their properties are in the center of interest for architecture evaluation, changes of these elements are focused by sensitivity analysis. In this approach, architecture changes are addressed in terms of changes and extensions of existing and promising architecture variants. Architectural decisions usually affect several parts of an architecture variant. Often, a single decision in terms of architectural changes causes complex effects regarding the architecture quality results. Moreover, even the architecture requirements can depend on the architecture, for example if restrictions are defined in relation to available resources specified by an architecture variant. As a basis for a consideration of these effects, the terms sensitivity and dependency need to be discussed in detail. On this basis, complex sensitivities can be investigated with respect to several dependencies of the evaluation result.

7.1.1 Dependency and Sensitivity

As introduced in Section 5.1, the sensitivity of architecture evaluation can be separated into structural and architectural impact. Structural impact, representing the relative importance of a partial evaluation result, will be taken into account to correlate sensitivities of partial evaluation results with respect to architecture changes. Architectural impact denotes the impact of an architectural change on partial evaluation result. For the discussion of dependencies, architectural impact is in the center of interest.

In the following sections, different kinds of dependencies are addressed with respect to architectural changes to explain which dependencies provide the basis for sensitivity. Sensitivity is not referred to as sensitivity point like in Bass et al. [BCK98] denoting a part of an architecture which may cause problems in achieving a particular quality goal. It is meant to be an explicit and quantified statement on how the quality of an architecture variant will be affected by architectural changes. Such changes can be directed to a single component and its properties as well as to more complex architectural decisions like e.g. the mapping of functions to controllers. In particular, changes of the mapping often have complex effects on several components of an architecture variant and thus its evaluation result.

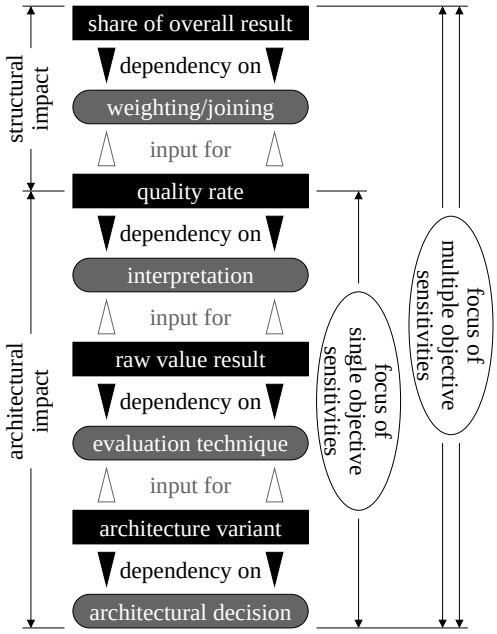

Figure 7.1: Dependencies in architecture development

Figure 7.1 depicts the dependencies on which the structural and architectural impact is based as well as the parts of architecture and evaluation which are contained

in the chain of dependencies. Sensitivities regarding a single objective—i.e. a single quality requirement/result—are based on the dependencies covered by the architectural impact. Sensitivities regarding multiple objectives—i.e. multiple qualities—are based on all of the dependencies represented in the figure.

7.1.2 Single Objective Sensitivity

According to Figure 7.2, three main questions arise regarding the dependencies on which single objective sensitivities are based.

- Which parts of the architecture are influenced by architectural changes?
  - Does an architectural change cause changes of element properties?
  - Does an architectural change cause changes of the mappings?
- Which elements are relevant for the evaluation technique?
  - Which properties of which element types are relevant?
  - Are available resources determined by the architecture?
- Which of the evaluation techniques subresults are relevant?
  - Will changes of a subresult have effect on the evaluation result?
  - Which elements are accountable for the effect?

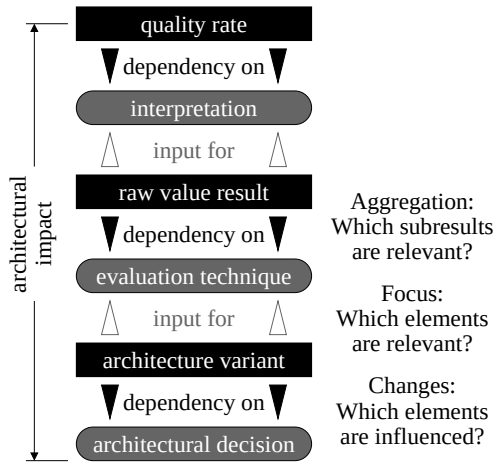

Figure 7.2: Dependencies regarding single objectives

In the following, some examples for sensitivities based on different dependencies are provided. The examples are motivated by the most common situations in architecture development in the automotive domain.

In the first example, the sensitivity of an architecture variant's quality is based on the dependency on *one specific property* of a set of components. The restrictions

specified by the quality requirement can be considered as independent with respect to an architecture in the this example which is the sensitivity of costs. The sensitivity is based on the costs property of each of the hardware components and the battery costs as well. The restrictions of costs actually are represented by the interpretation of the costs attribute. The effect of changes of the architecture in terms of using different or additional hardware may lead to changes of the costs of the architecture variant under consideration.

In the next case, the sensitivity is be based on *several different properties* of different types of components. An example is the sensitivity of the battery standby time. It depends on the power consumption of each of the controllers of the architecture. The restrictions of the resource—the battery capacity—are determined by the architecture variant and not by the quality attribute. Thus, this is an example for more than one dependency on the architecture.

Both of the sensitivity examples illustrated above are mapping-invariant. Actually, the properties of a controller, especially its power consumption, may change with changing function mapping. However, the overall power consumption, which is taken into account by the evaluation technique, will not differ very much as long as the controllers stay the same. The bottom line of the examples is that the resources are shared by the components without taking the mapping into account. The following sensitivities are directly affected by the mapping as the resources cannot be shared across components, e.g. controllers.

In the following example, the *mapping of functions to hardware* is taken into account. The sensitivity of performance quality attribute results usually does not only depend on component properties describing the resource capacities and those describing the resource consumption. The mapping of resource consuming components like functions to resource providing components like controllers cannot be taken into account as whole architecture. As the resources cannot be shared across components, the focus of the evaluation technique has to be considered. The architecture's resources do not only depend on its components. The availability of resources depends on the mapping of consumers as well. Thus, each of the resource providing components has to be considered in separate, which may lead to several dependencies to be analyzed.

In the next example, the *aggregation of subresults* of an evaluation technique are considered. According to the aggregation, the sensitivity has to be directed to the global effect of the change. If a local change does not affect the evaluation result, e.g. the worst utilization of some controller, the overall evaluation result will not be affected by this change either. Examples are the utilization of ROM, RAM, and CPU for which the resource capacity is given by and restricted to the respective components. A change of the function mapping leads to different utilizations which may have an effect on the quality or will let the quality be untouched if e.g. the highest utilization is not changed. An even more complex example is provided by the bus utilization which depends on the communication mapping that in turn depends on

the function mapping. Moreover, the utilization is not necessarily linear. In the case of bus utilization, it depends on the bus protocol. Thus, small changes may have strong effects.

Especially for the latter sensitivities, these can only be determined for concrete architecture variants and concrete changes. Hence, analyzing architecture based on such dependencies can be quite expensive and is no appropriate means for automatically improving arbitrary architecture variants. A thoughtful selection of architecture variants and promising changes is still essential for architecture analysis.

7.1.3 Multiple Objective Sensitivity

Usually, more than one quality attribute result is affected by an architecture change. This depends on the architecture parts which are affected and the dependencies on these parts on which sensitivities are based. These sensitivities can become quite complex as presented above. In this section, sensitivities are assumed to be known. The main attention is directed to interdependencies between sensitivities.

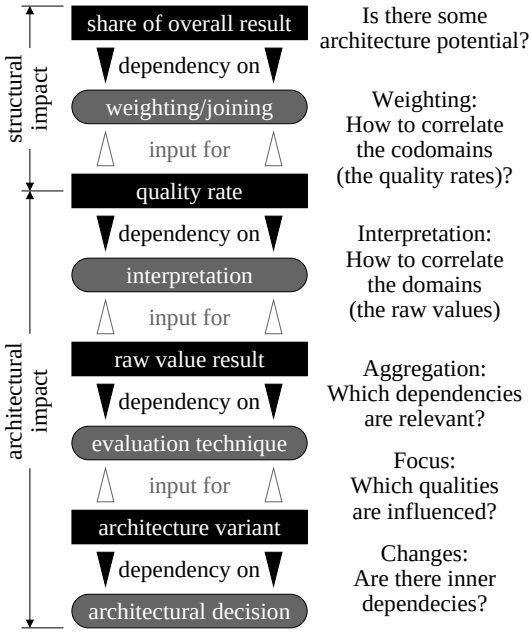

Figure 7.3: Dependencies regarding multiple objectives

If two qualities depend on the same parts of an architecture to be changed, conflicts in sensitivities can arise. In case that a change has a more or less positive effect on one of the qualities and a more or less negative effect on the other one, the respective sensitivities are called conflictive. Bass et al. [BCK98] call such situations for an architecture a tradeoff point. But not only the existence of the tradeoff but

rather its explicit consideration based on the explicitly expressed sensitivities is most interesting for decision support regarding changes, i.e. their direction and magnitude.

Figure 7.3 picks up and extends the set of questions regarding single objective sensitivity asked at the beginning of Section 7.1.2. At this point, the most interesting question is directed to inner dependencies of/between architecture elements influenced by changes of the architecture. These inner dependencies are discussed in the following paragraphs and will be the basis for later correlation of conflictive sensitivities addressed in Section 7.3.2.

There are three classes of inner dependencies. The first class deals with sensitivities based on the same component properties of the architecture. The second one is concerned with dependencies on the same components but not the same properties of the component. The third class covers conflictive sensitivities with inner dependencies which are not directly based on specific components but on e.g. the function mapping. A change of the mapping usually affects many parts of an architecture variant.

For the first class, no additional dependency needs to be taken into account as the sensitivities are already based on the same parts of the architecture. Thus, they can directly be correlated taking the structural impact into account.

For the second class, a strong relation between the properties of the components on which the sensitivities are based can be taken for granted. This relation needs to be expressed in order to be able to correlate the sensitivities. An example is the relation between processor speed and controller costs in the In-Car Radio Navigation case study. Changes of a controller will lead to changes of its costs as well as of its processor power.

For the third class, no general advice how to obtain the relation between the architecture parts on which the sensitivities are based can be provided as multiple different components may be involved. It strongly depends on the underlying architecture and the sensitivities themselves. However, this third class is an example for so-called expert knowledge in the field of sensitivity analysis. Even if concrete relations cannot be expressed, tendencies can be described by experts and can be taken into account for architecture development.

For Architecture Potential Analysis as presented in Section 7.3, the relations—or inner dependencies—can be accounted to the second class. Even several relations in a chain may become necessary to express the inner dependencies. However, the second class can still be handled while dependencies of the third class usually are too complex for a precise consideration. Dependent on the complexity of an architectural change, such dependencies may be reduced to class 2 dependencies by expert knowledge providing development tendencies. Unfortunately, tradeoffs of the first class are mostly rare.

7.2 The Modular Performance Analysis

One of the main quality attributes in the embedded system domain represents performance requirements. The Modular Performance Analysis (MPA) by Wandeler et al. [WTVL06, WTVL04] has been developed to evaluate embedded system performance based on performance models of embedded system components. For performance evaluation of the In-Car Radio Navigation System case study, the MPA has been applied (see Section 6.2). Besides using the analysis results as partial evaluation results in the QADAG instance of the case study, the MPA can be used for design space exploration. Not only the performance of the currently evaluated architecture variant but rather of variants after changes can be assessed. Inner dependencies of architecture variants can be uncovered and used as input for Architecture Potential Analysis as introduced in Section 7.3. The principles of the MPA are presented in the current section.

7.2.1 Real-Time Calculus

The theoretical background of the MPA is based on Real-Time Calculus by Thiele et al. [TCN00, TW04]. The Real-Time Calculus is directed to networks of computation resources. Each of the resources in the network provides a specific capacity for computation or communication of events (e.g. user request, messages, interrupts). Information on the occurrence of the events as event streams and the resource capacity and time needed for processing the events build the basis for calculation of the delay between event occurrence and its complete processing.

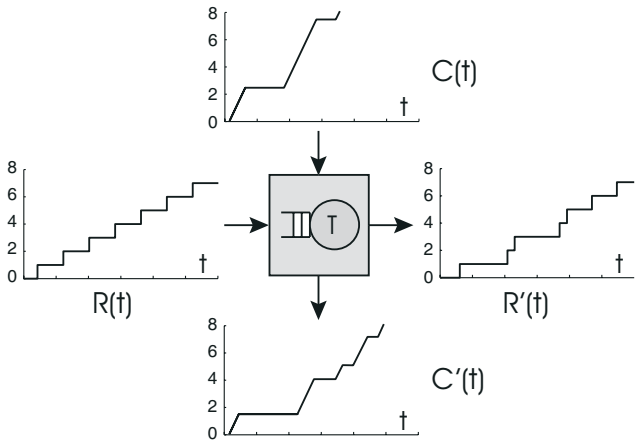

Figure 7.4: MPA event processing, see [WTVL06]

Figure 7.4 shows an example for a resource which processes events. The function $R(t)$ (left hand side) describes the occurrence of events over time. It is called the

request function. The function $R'(t)$ (right hand side) describes the event stream after its processing and will be the input for the next resource to process the event streams. Delays caused by the processing by this resource are contained in function $R'(t)$. It is called the *delivered computation function*. The *capacity function* $C(t)$ (at the top) describes the capacity of the resource. It is the maximum of capacity amount that could be delivered up to time t by the resource. The capacity remaining after processing the events is represented by function $C'(t)$ (at the bottom) which is called the *remaining capacity function*. In a network of resources, the delivered computation function as output of one resource will be the request function of another resource. The remaining capacity function as output of one resource will be the input for the same resource for another event stream to be processed. Hence, the processing delay for an event can be calculated with respect to all resources needed to process the event considering further events currently processed by the network. How to build the network on basis of architecture models and scenarios will be presented later in this section.

7.2.2 Timed Automata

For calculating the delay caused by a single resource, a model describing the temporal behavior of the resource becomes necessary. For modeling the temporal behavior, timed automata are used which extend automata by clocks (dense time) and clock constraints. The state of a timed automaton is represented by a location and a clock interpretation. A transition of a timed automaton is either a switch (a transition between locations) or a change in time. Furthermore, timed automata can be synchronized via channels which will be needed to build the resource networks mentioned above. For a detailed introduction on timed automata, see Alur and Dill [AD94]. The implementation of timed automata in the model checker UPPAAL is used as reference tool in the MPA. An introduction on UPPAAL is given by Behrmann et al. [BDL04]. Templates for distributed system architectures are provided by Perathoner et al. [PWT05]. In the following, examples of timed automata for the architectures analyzed in Wandeler et al. [WTVL06, WTVL04] are presented. These examples are taken from Hendricks and Verhoef [HV06].

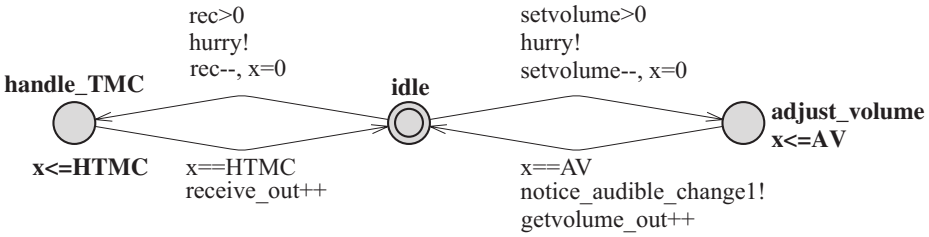

Figure 7.5: Radio function automaton, see [HV06]

Figure 7.5 depicts a timed automaton for the behavior of the radio function handling TMC (Traffic Message Channel) messages and adjusting the volume. $HTMC$ and AV denote the time needed for handling a TMC message and adjusting the volume, respectively. Obviously, the radio function can either handle a TMC message or change the volume but not both at the same time.

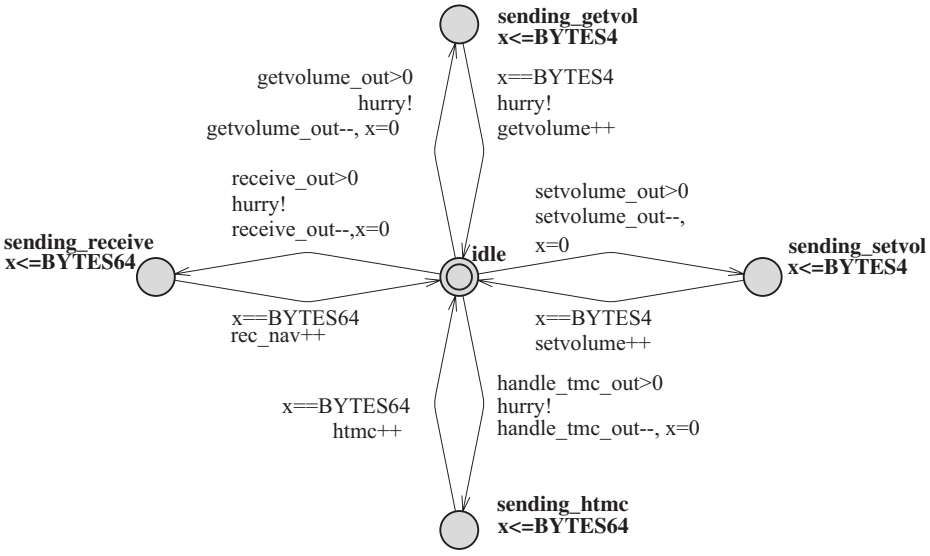

Figure 7.6: Bus automaton, see [HV06]

Figure 7.6 contains a timed automaton for the bus of the case study in Wandeler et al. [WTVL06]. The time needed for processing (sending) an event depends on its size. Therefore, $BYTES64$ and $BYTES4$ are declared to be used for modeling the sending of various events.

Automata as shown above can be used to model the temporal behavior of architecture components. The processing time can be expressed by such models to calculate the delay of event processing. Temporal logic expressions (see Clarke et al. [CES86]) can be used to express requirements to be checked by the UPPAAL model checker. Even upper and lower bounds of processing delays can be calculated.

7.2.3 Modular Performance Analysis Model

The system structure on which the MPA can be performed is built of resource models containing timed automata. Scenarios describing user interaction can be given as sequence diagrams as shown in Figures 4.6, 6.13, and 7.7. They determine which functions need to communicate with each other in order to process an event. An architecture model is necessary to determine by which resources an event is processed on its way through the system. The architecture model presented in Chapter 2 with

its function mapping and communication mapping provides the required views. From the architecture model and scenarios, a Modular Performance Analysis model can be derived building the resource network on which the MPA can be performed.

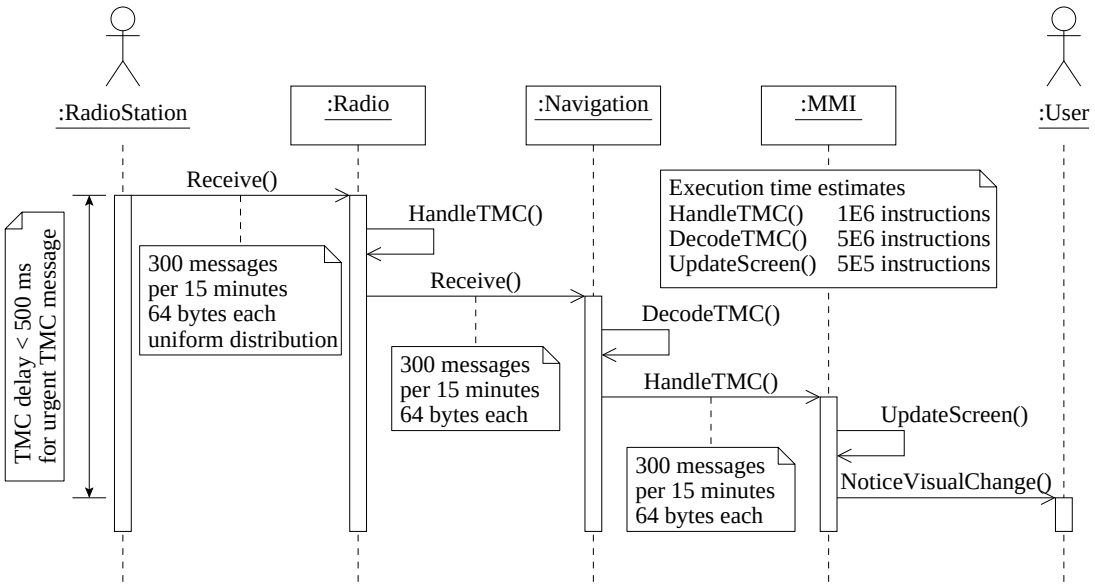

Figure 7.7: TMC message handling scenario, cf. [WTVL06]

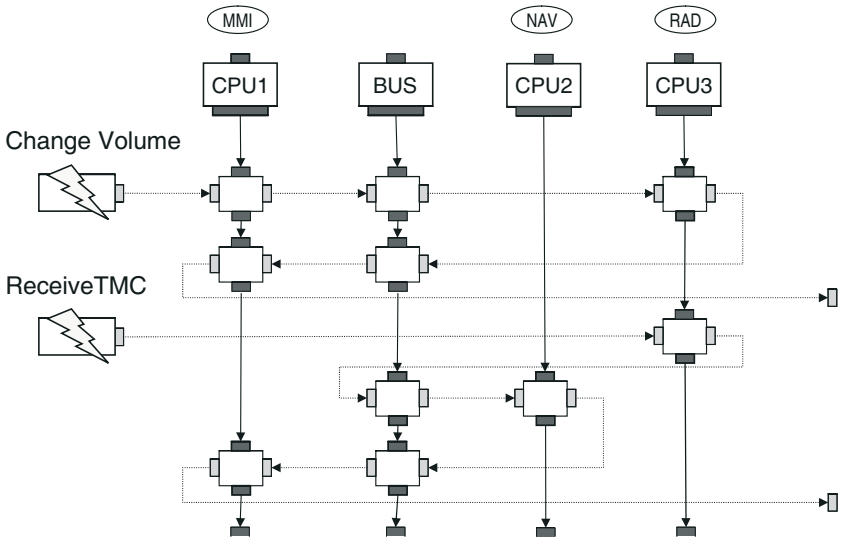

Figure 7.8: MPA model, see [WTVL06]

Figure 7.8 depicts such a performance model containing two scenarios. At the top of the figure, the function mapping can be seen. The mostly horizontal arrows describe the event streams' ways through the system resources for processing the events (cf. Figure 7.4). In terms of scenarios, the event can be considered as the stimulus. The vertical arrows describe the order of resource utilization.

Performance models usually are quite expensive to be constructed and to be adapted to an architecture variant. In case of the performance modeling used in the MPA, the models are build highly modularly. For additional architecture variants with changed function mapping, an MPA model can be provided quite efficiently. Available performance models of the resources can be reused to build the resource network on which the MPA model is based. This is a big advantage in terms of efficiently performing architecture evaluation and even design space exploration regarding system performance.

7.2.4 Modular Performance Analysis Results

As stated above, the MPA is capable of providing results in terms of design space exploration. By changing e.g. the processing capacity of a CPU, the capacity for event processing of functions mapped to the CPU will be increased and the delay of event processing will be decreased. Actually, no additional models or new scenarios are needed for such considerations. The MPA, as implementation of the Real-Time Calculus, can calculate the effect of such changes. An output regarding possible changes of processor speed is given in Figure 7.9. The analysis does not only provide a single evaluation result with respect to a single architecture but rather provides results on changes of an architecture variant. Such results will be input for the Architecture Potential Analysis which will be introduced in Section 7.3.

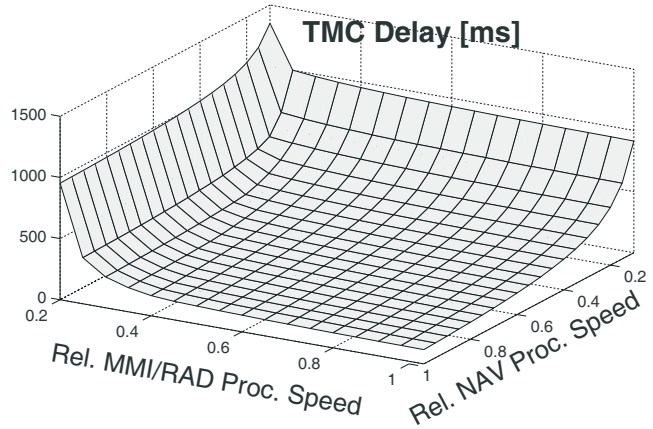

Figure 7.9: MPA result (Variant II), see [WTVL06]

7.3 Architecture Potential Analysis

In this section, Architecture Potential Analysis will be introduced as first defined by Florentz [Flo07b]. It is motivated by the need for knowledge of the insides of the evaluation, i.e. sensitivities of the architecture quality. With this knowledge, architecture space exploration can be performed more efficiently. Without this knowledge, the evaluation of a changed architecture variant has to be performed to check its quality, which can be quite expensive. Known sensitivities can guide the development and changes of an architecture and save development resources. Furthermore, they help to explain and document architectural decisions. Besides for communication purpose, documentation is quite important because sensitivities may change over time. For example, the tradeoff between costs and performance depends on the market and available technology. Over the years, the costs for performance (i.e. powerful hardware) will decrease as well as the willingness to pay will do. But the need for performance will rise in order to provide additional functionality. This is just a small example for complex sensitivities. To actually analyze architecture and its evaluation regarding its potential, i.e. the possibility of increasing the overall quality of an architecture variant, the sensitivities have to be taken into account seriously.

The identification of relevant—to be analyzed—sensitivities is addressed in Section 7.3.1. In Section 7.3.2, the principles of the application of sensitivities in Architecture Potential Analysis are presented.

7.3.1 Identifying Relevant Sensitivities

Most of the quality attributes depend on a set of properties of the architecture components as well as on the architecture decisions—especially the mapping of functions to controllers—and their changes. Neither all of the sensitivities are known nor all of them are relevant for further analysis. For performing Architecture Potential Analysis in order to identify potential of selected architecture variants, i.e. regarding a single decision or a small set of decisions, promising sensitivities or even tradeoffs have to be identified. The first analysis technique presented in this section is the result screening to be applied to a complete set of architecture variants (at least with respect to a certain set of represented decisions). The second technique refers to expert knowledge and tradeoff analysis as commonly known.

Result Screening

Identifying relevant sensitivities by screening the results requires full evaluation of all variants based on a set of decisions. For example, in the Body Comfort System, these decisions concern the mapping of the Short Lift Control (2 variants), the realization of the Convertible Top Control (3 variants), and the battery capacity (3 variants), which leads to a total of 18 architecture variants as presented in Section 3.1. In

order to support a result based screening, the results need to be ranked as shown in Table 6.6 for the Body Comfort System case study. K.O. results are separately ranked and appended to the list.

The first step of the screening is a grouping of the ranked results. Groups are motivated by similarities of decision variants, by order patterns of decision variants in the ranking, by the quality result differences in ranked variants, and by K.O. results. Grouping helps to structure the results in order to reduce the number of relevant variants to be compared. A pairwise screening of results leads to a number of $\binom{n}{2}$ comparisons which means 153 comparisons for 18 variants. This is not only costly to perform but rather bears quite a great number of comparison results to be interpreted. The interpretation or even investigation of dependencies identified during the grouping is the second step of the result screening and will be addressed below.

rank	**variant**			**pattern**
	A	B	C	
1	A1	B1	C1	Decision C has more impact than Decision B
2	A1	B2	C1	
3	A1	B1	C2	
4	A1	B2	C2	
5	A2	B1	C1	Decision B has more impact than Decisions C
6	A2	B1	C2	
7	A2	B2	C1	
8	A2	B2	C2	

Table 7.1: Order patterns for Decisions B and C

An abstract but simple example can be given based on three decisions, namely A, B, and C, with two decision variants each. Table 7.1 contains a ranking of all architecture variants. The first four and second four variants have similarities of A decision variants. But depending on the actual decision variant of A, the ordering of Decisions B and C changes. For A1, Decision C has more impact then Decision B. B1 leads to better results than B2 and C1 than C2. Variant A1B2C1 is better than A1B1C2, i.e. changes of C have more impact. The sensitivity difference will change if Variant A2 is selected. Both of the A2B1 results are better than the A2B2 results regardless Decision C.

Similarities of variants regarding one or more decisions means to mask the decisions' impact on the results. The ranking of results is not influenced by this masking of decisions. Thus, the sensitivity of the varying (not masked) decisions can be highlighted and compared. Actually, the masking of arbitrary decisions and their influence can simply be achieved by sorting the variants by these decisions. The

result ranking should be kept up where possible although the decision-based sorting has priority over the result ranking in this case.

For applying the grouping, not only the overall ranking should be kept up but rather differences in the quality results should be observed. After all, those sensitivities are most relevant which may provide hints how and why some quality has been achieved. Thus, big quality differences motivate a separation of the respective variants into different groups. An example for a proper grouping is provided by the Body Comfort System case study in Section 8.1.

The similarities of grouped variants provides a representative-based inter group comparison which drastically decreases the effort. Furthermore, differences of inner group dependencies based on order patterns can be investigated. The second step of the result screening is the investigation of dependencies identified during the grouping. By starting with the quality results, only the existence of dependencies is observed. Further knowledge of these dependencies is not yet available. Thus, experts have to investigate the dependencies based on their knowledge, which is a quite informal process. Their insights can be used to explain why some architecture variant is to be preferred over another one and which architecture decision variants are to be preferred in future development. In most cases, the result screening is more reasoning support than decision support. However, if sufficient modeling details are available, even tradeoff analysis like presented in the following section can be performed. Hence, result screening can be considered as starting point for further analysis. In such cases, additional modeling details may be required for a precise investigation of the dependencies.

Tradeoff Analysis

Architecture evaluation is often performed for only a small set of possible architectures. First, short development resources restrict the number of variants to be taken into account. Second, only a few of them are promising anyway. Such variants need to be identified based on experiences and expert knowledge and are often derived from legacy systems. Because of the relatively small set of architecture variants, complete coverage even of only some decisions is quite unlikely. Thus, result screening is not the first choice for identifying relevant dependencies in such cases. But the limited number of variants often enables more detailed evaluation and investigation, in particular with respect to tradeoffs. Moreover, relevant dependencies leading to tradeoffs may already be commonly known in an application domain or can be identified by tradeoff analysis (e.g. the Architecture Tradeoff Analysis Method, ATAM, see Bass et al. [BCK98] and Clements et al. [CKK01]). Even result screening can be used to identify tradeoff but requires a complete set of architecture variants as stated above. Unfortunately, this cannot be taken for granted.

The performance models in combination with the relatively flexible resource allocation of the MPA provides insights of architecture dependencies which are relevant for

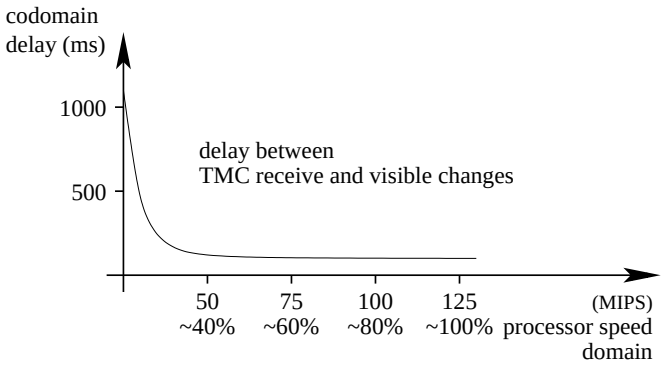

Figure 7.10: TMC delay versus MMI processor speed, cf. [WTVL06], Fig. 18

tradeoff analysis. The In-Car Radio Navigation System is evaluated, amongst other techniques, by the Modular Performance Analysis (see Section 6.2). Figure 7.10 (cf. Wandeler et al. [WTVL06], Fig. 18) contains the reaction delay regarding the TMC message handling scenario (see Figure 7.7) against the processor speed of one of the processors, on which the man machine interface is mapped. It is an extract from the MPA result presented in Figure 7.9. The dependency of reaction delay on processor speed is used for analysis of the In-Car Radio Navigation System in Section 8.2.

7.3.2 Quality Correlation

Architecture quality is represented by quality attributes and eventually expressed in terms of quality rates. Quality rates, after all, depend on the evaluation results in terms of their interpretation (see Section 4.2.4). Thus, an interpretation represents the dependency of the quality rate on the evaluation result. Those interpretations need to be correlated in order to express tradeoffs between architecture qualities. The knowledge about tradeoffs as well as the achievements of result screening can now be used to identify further dependencies which may be relevant—even necessary—for architecture requirements correlation. The correlation is performed in two basic steps. The first one regards domains of dependencies and thus takes the values into account on which some other values depend. The second one regards the codomains of dependencies and thus takes the values into account which depend on another ones. Figure 7.10 shows a dependency of the *codomain delay in ms* on the *domain processor speed in MIPS*. Taking interpretations as an example, the domain is given by the evaluation result measurement. The codomain is given by the quality rate measurement.

The correlation of two domains will require additional knowledge on inner dependencies if the domains are not the same. Even more, the domain correlation can be based on several correlations itself. But, those are principally step-by-step corre-

lations of different domains. Figure 7.11 contains two simple examples for domain correlations. Figure 7.12 depicts a correlation chain over three domains.

The codomain correlation is considerably easier to perform. As the codomains to be correlated are architecture requirements expressed by quality rates, they already are alike. Thus, no different codomains have to be brought into relation. The correlation of the codomains is simply based on the structural impact, i.e. the absolute weight of the involved quality attributes representing the requirements. By building the weighted sum over the correlated domains, the partial result (regarding two quality attributes) can be calculated.

Figure 7.13 depicts the potential of an architecture variant based on the correlation of the quality rates of two quality attributes. The joined result contains the codomain correlation with respect to the different weights of the quality attributes.

In Section 8.2, the correlation of quality attributes is performed on basis of an MPA result (see above). Two special cases for correlation will be included in the case study. First, one of the correlation steps regarding the domains requires an adjustment because the requirement domain is a relative expression regarding different requirements whereas the dependency domain is given as absolute values. The rela-

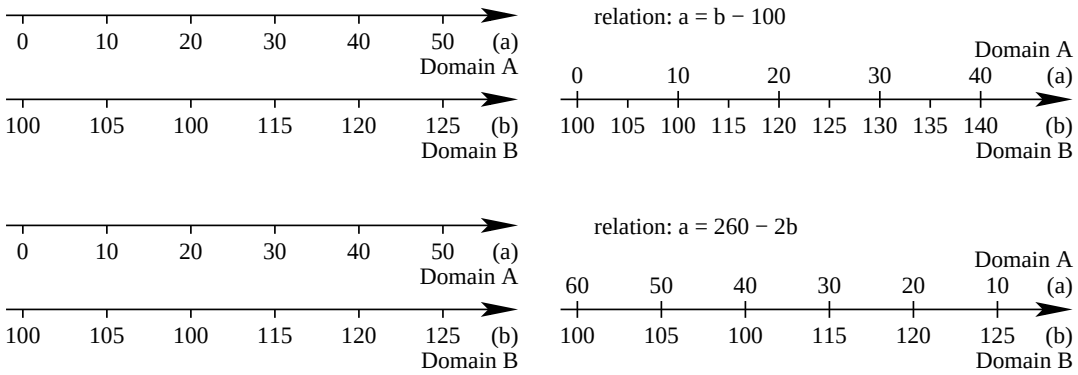

Figure 7.11: Correlation examples

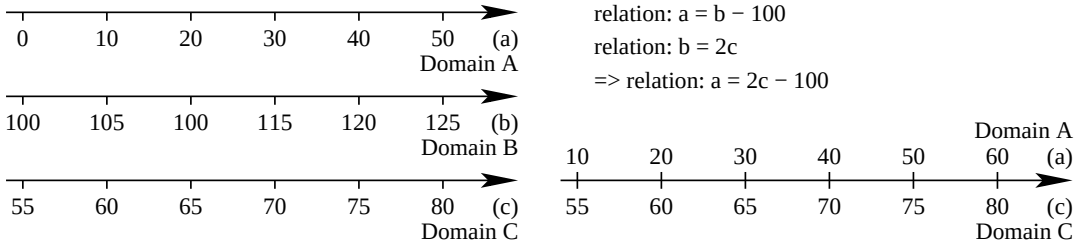

Figure 7.12: Correlation chain over three domains

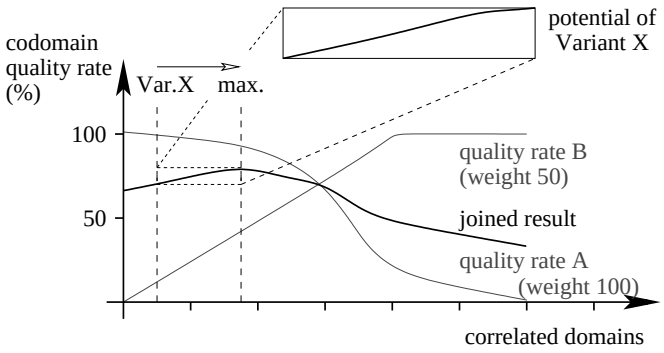

Figure 7.13: Correlation of quality interpretations

tive expression is necessary in order to cover use cases with different requirements, i.e. different expected and tolerated delays. However, as the absolute requirements for each of the use cases are known, they can be adjusted to match the relative expression. Second, if there are significant differences (offsets) between e.g. two architecture variants, which are not covered by the sensitivities, they can be integrated by shifting the correlation. This leads to several correlation results according to the offsets. An example is a dependency between costs and performance. Only a part of the costs is taken into account by the cost measure of a dependency but some of the architecture variants contain additional costs. The additional costs—actually cost differences—can be considered by shifts in the correlation as shown for the network costs of the In-Car Radio Navigation System case study.

7.4 Related Work – Architecture Analysis

The Cost Benefit Analysis Method as one of the most common approaches taking the overall architecture into account will be presented and compared to the Architecture Potential Analysis. The intention of the approaches and the way to achieve architecture improvement are in the center of interest in this discussion.

The Cost Benefit Analysis Method

With respect to an efficient development, the Cost Benefit Analysis Method (CBAM) by Kazman et al. [KAK01, KAK02] will append the ATAM process. The scenarios prioritized in the ATAM process and maybe newly identified ones will be used to select architecture strategies leading to the highest benefit under consideration of costs and time. As depicted in Figure 7.14 (cf. Kazman et al. [KAK01, KAK02]) costs are meant to be development costs and not unit costs like in the case studies of

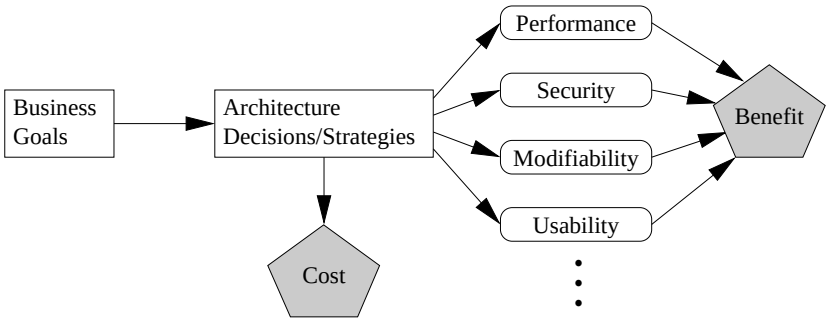

Figure 7.14: Context for the CBAM, cf. [KAK01, KAK02]

this work. They are not represented as quality attribute but build an upper bound for ongoing development. The steps of CBAM are briefly presented in the following.

Step 1: Collate Scenarios
The scenarios identified and prioritized during the ATAM process are reviewed and new ones may be added (and prioritized as well) by stakeholders. The top one-third will be considered in the ongoing CBAM.

Step 2: Refine Scenarios
If not yet done, the stimulus and response measure of the scenarios is refined. The response measure will be mapped to response levels of *worst*, *current*, *desired*, and *best-case*.

Step 3: Prioritize Scenarios
100 votes will be allocated to each of the stakeholders. The votes will be distributed among the scenarios with respect to the desired response level. The top 50 % of the scenarios according to their total of votes will be selected. The highest ranked one gets a weight of 1.0 assigned. The others are weighted in relation to their votes. This weights will be needed for calculating the overall benefit of an architecture decision/strategy.

Step 4: Assign Utility
The utility of the response level will be assigned for all scenarios.

Step 5: Develop Architectural Strategies for Scenarios and Determine Their Expected QA Response Levels
With respect to the scenarios identified in Step 3, architecture strategies are developed and already developed ones are taken into account. The response level, that is expected to be achieved by implementing the strategies, will be determined.

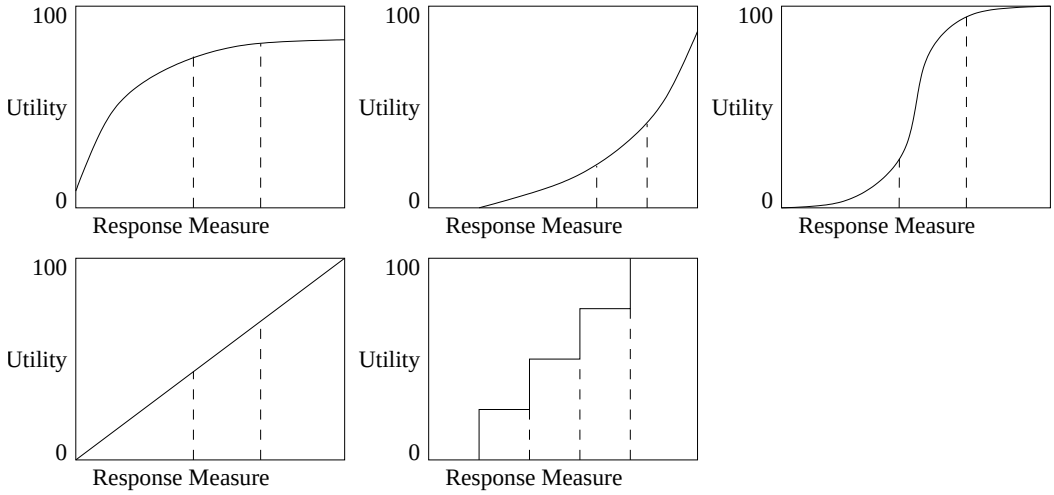

Figure 7.15: Utility response curves examples, cf. [KAK02]

Step 6: Determine the Utility of the Expected Quality Attribute Response Level by Interpolation
For each of the scenarios, the utility of the expected response level will be determined. The relation between utility and response measure is represented by so-called utility response curves as depicted in Figure 7.15.

Step 7: Calculate the Total Benefit Obtained from an Architectural Strategy
The utility difference of the current and the expected level will be normalized using the weights determined in Step 3. On this basis, the benefit of architectural changes/strategies can be calculated.

Step 8: Choose Architectural Strategies Based on ROI Subject to Cost and Schedule Constraints
With respect to cost and schedule constraints, the architecture strategies leading to the highest return on investment (ROI), i.e. the ratio between benefit and costs, will be chosen. This is done by ranking the strategies based on their ROI. The top strategies will be implemented until the budget or the schedule are exhausted. Thus, the most efficient strategies are implemented.

Step 9: Confirm the Results with Intuition
A confirmation between the results and the business goals will be done. If significant issues seem to be overlooked, an iteration of the steps starting at Step 2 will be performed.

The CBAM considers one architecture to be developed. The architecture strategies identified by the CBAM to be implemented are meant to achieve the highest

benefit with respect to the available development resources. In terms of architecture evaluation of a set of variants, this is the most promising architecture variant. But in contrast to the case studies of this work, costs are not considered as quality attribute but as short development resource that will be exhausted by the most promising architecture strategies. Thus, not a selection of architecture variants but a selection of strategies regarding one architecture is the intention of CBAM. Nevertheless, the CBAM can be compared to the Architecture Potential Analysis presented in the current chapter as the latter is meant to identify promising changes (or even further development) of selected architecture variants as well. Both approaches can be considered as some kind of architecture space exploration. The similarity of the utility curves of CBAM and the interpretation instances of the QADAG underline common principle ideas. The main difference in exploring the architecture space is that the CBAM identifies architecture strategies, i.e. a set of architecture changes or further development, and chooses the most efficient ones to be realized with limited development resources. The Architecture Potential Analysis works the other way around. Without directly taking development effort into account, the most promising changes are identified on the basis of dependencies between architecture and the evaluation results, i.e. the achieved quality. The explicit modeling of architecture requirements as QADAG instance and the consideration of concrete hardware as part of the architecture provide this technical background and enable explicit and quantified tradeoff consideration.

7.4.1 The SymTA/S Approach

An alternative to the Modular Performance Analysis is the SymTA/S approach by Henia et al. [HHJ+05]. It has been used in combination with the QADAG as well (see Florentz et al. [FGB+07]). Like MPA, SymTA/S has already been applied in the automotive domain (see Richter and Ernst [RE06]). SymTA/S implements timing analysis on process and task-level models. Single controllers as well as controller networks can be analyzed regarding their time behavior and possible problems and solution can be identified. Although it has not been used on the case studies presented in this work, application fields of the SymTA/S approach with respect to architecture evaluation as presented in this work will be outlined in the following.

Design space exploration based on the SymTA/S approach developed by Hamann et al. [HRE06, HE07] provides limitations of changes of resource usage in order to avoid performance lacks and point out safety margins of resources. Information like this can be used to identify insufficient resource utilization and avoid unfavorable changes. Moreover, relations between performance and other extra-functional qualities can be investigated as starting point for tradeoff analysis and thus as input for Architecture Potential Analysis. In return, Architecture Potential Analysis can help to motivate the investigation of specific parts of an architecture by design space exploration based on the SymTA/S approach. First, a selection of promising archi-

tecture variants is provided by architecture evaluation. Second, relevant (and maybe critical) parts of an architecture variant can be identified by Architecture Potential Analysis. Iterations between Architecture Potential Analysis and design space exploration based on SymTA/S can help to achieve efficient usage of development resources. Relevant parts can be identified, investigated, and after all be fed back to the evaluation process.

However, the SymTA/S approach requires models on task-level which are hardly available for early evaluation but will become necessary for later design anyway. Thus, for late evaluation and precise adjustments of an architecture variant, the SymTA/S approach is very appropriate. Two main application fields of the SymTA/S approach in late evaluation will be addressed in the following. Late evaluation usually is performed to assure the quality of one or at most a small set of architecture variants. With growing availability of modeling details, information on tasks and communication including execution and transmission priorities become available as well. The SymTA/S approach can be applied to assure that latencies are kept low to avoid too big action-reaction delays. Moreover, changes of task execution and communication scheduling can be identified in order to decrease such latencies and avoid exceeding deadlines. This is the second case of application in late evaluation. If an overall promising architecture variant fails regarding one performance quality, SymTA/S may suggest a solution to fix the performance problem. Thus, an otherwise insufficient variant will get another chance if the architecture variant's overall result is worth to be considered some further.

8 Analysis Case Studies

The case studies evaluated in Chapter 6 are analyzed in this chapter regarding their evaluation results. The Body Comfort System case study is an example for a complete evaluation of all architecture variants regarding the architecture decisions to be made. Relevant sensitivities are identified by result screening. Grouping of the results will help to keep the analysis effort low. The analysis results will help to understand why some architecture variants are to be preferred over others. Moreover, hints for future development and even promising changes of architecture components can be provided. In this case study, changes of the function mapping are not to be expected anymore as the architecture variants completely cover all mapping variants to be considered regarding Short Lift Control and Convertible Top Control. In the second case study, the In-Car Radio Navigation System, results of the Modular Performance Analysis are part of its evaluation. This case study covers only a small subset of possible—and promising—variants. The Modular Performance Analysis provides further results, which represent inner dependencies of the evaluated architecture variants. Here, they can be used to estimate the impact of changes to identify unused architecture potential.

In Section 8.1, the Body Comfort System case study is analyzed. The In-Car Radio Navigation System case study is analyzed in Section 8.2.

8.1 Body Comfort System Analysis

The Body Comfort System case study contains 18 architecture variants. All of them have been evaluated in Section 6.1. With respect to the three decisions taken into account, the 18 architecture variants build a complete set as well as the 18 evaluation results do. Hence, analyzing this case study will not identify new architecture variants in terms of the three decisions. The benefit of performing the analysis is directed to explain why some architecture variant is ranked higher than others. What can be done to improve (partial) results and what can be learned from the overall evaluation?

At the end of Section 6.1, four architecture variants have been stated to be quite interesting with respect to the architecture decisions which they do represent. The decision results can be interpreted on basis of the evaluation results. The impact of decisions on the ranking is more complicated to explain. In the following sections, a result screening is performed to identify relevant sensitivities leading to the result ranking. On that basis, selected architecture variants can be compared to get an idea of the reasons on their ranking. Afterwards new insights regarding inner dependencies are discussed.

8.1.1 Body Comfort System Result Screening

The screening will be performed in several steps according to Section 7.3.1. First, reasons for K.O. result are analyzed in Section 8.1.1. Second, ordering patterns are identified with respect to similarities of decision variants in the ranking in Section 8.1.1. Finally, result groups are built based on the previous considerations.

K.O. Reasons

Besides the K.O., the six worst evaluation results have a similarity in the decision regarding the Convertible Top Control realization. All of those—and only those—results have the control realized by the Reduced Convertible Top Device. And all of them are rejected because of too high costs that actually are caused by the quite expensive Reduced Convertible Top Device.

Three of the architecture variants are evaluated to K.O. because of too less standby time. They even have double similarity, one regarding the battery capacity (50 Ah) and another one regarding the Short Lift Control mapping (to the Power Window Devices). Actually, it is the combination of applying the extended Power Window Devices with Short Lift Control and providing less battery capacity.

Although the Reduced Convertible Top Device is not the only hardware component causing costs, the similarity of the set of the architecture variants that are K.O. because of too high costs identifies this device as one of the main reasons. Especially the fact that none of the architecture variants containing the Reduced Convertible Top Device succeeded in the evaluation consolidated this suspicion. The K.O. caused by too less standby time is an example for a combination of decisions accountable for this result. Successful architecture variants may at least partially represent the same decisions even if not in combination. For later sorting of the architecture variants to mask certain decisions, the K.O. of the results should be omitted to enable a proper sorting. This will be no problem for the aspired grouping because the sorting is only a temporal means and will be undone afterwards and as the K.O. cannot necessarily be accounted to one decision it should not be taken into account for sorting as well. Usually, the K.O. results have a lower quality assigned anyway because a K.O. means to have 0 % quality for at least the attribute for which the K.O. has been evaluated.

Order patterns

Omitting the architecture variants containing the Reduced Convertible Top Device, which actually are K.O., the attention is directed to the architecture variants with Convertible Top Control on the Mini Convertible Top Devices or the Body Control Device. Sorting the architecture variants according to the Short Lift Control mapping reveals order patterns for the Power Window Device as well as for the Body Control Device for the first four architecture variants, respectively. The similarity of Power Window Device decisions contains an order of the Convertible Top Control whereas the Body Control Device decisions contain an order regarding the batter capacity.

Actually, this order patterns are directly represented in the ranking, which makes it even more obvious that there is some interesting and maybe relevant sensitivity.

rank	**result**	**variant**		
		SLC	CTC	battery
		common	**sorted**	**unsorted**
1	70.6 %	PWD	MiniCTD	61 Ah
2	69.6 %	PWD	MiniCTD	72 Ah
3	67.3 %	PWD	BCD	61 Ah
4	65.3 %	PWD	BCD	72 Ah
		common	**unsorted**	**sorted**
5	61.5 %	BCD	MiniCTD	61 Ah
6	60.1 %	BCD	BCD	61 Ah
7	59.5 %	BCD	MiniCTD	72 Ah
8	58.0 %	BCD	BCM	72 Ah

Table 8.1: BCS evaluation result patterns

Table 8.1 extracts the eight best results of the ranking of Table 6.6. The ranking is completely kept up. The architecture variants with 50 Ah battery capacity do not match the observed patterns and are not contained in the table. In accordance to the ranking, they have a similarity regarding the battery capacity. Because of the K.O. results for the Short Lift Control mapping, the order of that decision is changed in contrast to the first eight architecture variants.

Variant Groups

Motivated by the order patterns, the first and the second four highest ranked architecture variants are grouped, respectively. The third four architecture variants have a similarity as stated above but are separated by the K.O. results. Thus, two groups are built for them. The last group is based on the K.O. results caused by the costs of the Reduced Convertible Top Device. Moreover, the differences between the last and first result of two neighboring groups consolidate the grouping as they are bigger than the differences of neighboring results in the groups. Following observations can be made regarding the grouping shown in Table 8.2. They will be discussed in the next section.

- The order of the middle architecture variants of Groups A and B are swapped according to the represented decisions. A change of the Short Lift Control mapping seems to have far-reaching effects revealing sensitivities to be taken into account.

- Besides the K.O. of the architecture variants of Group D, their result seems to be promising compared to Groups A and B. The K.O. even seems to avoid the group's positioning in between of Groups A and B despite a small difference of the results.
- Group E contains battery capacity orders from the biggest to the smallest capacity. Although the respective architecture variants are not promising, this observation may help to understand inner dependencies.

rank	**result**	Δ	**variant**		
			SLC	CTC	battery
Group A				Group A	
1	70.6 %	1.0 %	PWD	MiniCTD	61 Ah
2	69.6 %	**2.3 %**	PWD	MiniCTD	72 Ah
3	67.3 %	2.0 %	PWD	BCD	61 Ah
4	65.3 %		PWD	BCD	72 Ah
Group B		3.8 %		Group B	
5	61.5 %	1.4 %	BCD	MiniCTD	61 Ah
6	60.1 %	**0.6 %**	BCD	BCD	61 Ah
7	59.5 %	1.5 %	BCD	MiniCTD	72 Ah
8	58.0 %		BCD	BCM	72 Ah
Group C		3.4 %		Group C	
9	54.6 %	2.1 %	BCD	MiniCTD	50 Ah
10	52.5 %		BCD	BCD	50 Ah
Group D				Group D	
11	63.4 %	(K.O.)	PWD	MiniCTD	50 Ah
12	60.2 %	(K.O.)	PWD	BCD	50 Ah
Group E				Group E	
13	56.0 %	(K.O.)	PWD	redCTC	72 Ah
14	55.9 %	(K.O.)	PWD	redCTC	61 Ah
15	47.6 %	(K.O.)	PWD	redCTC	50 Ah
16	42.3 %	(K.O.)	BCD	redCTC	72 Ah
17	42.2 %	(K.O.)	BCD	redCTC	61 Ah
18	34.0 %	(K.O.)	BCD	redCTC	50 Ah

Table 8.2: BCS evaluation result groups

8.1.2 Body Comfort System Insights

Groups A and B The change of the order in Group A and Group B is a hint for relevant sensitivities based on the Short Lift Control mapping. Especially, the differences between the results of the 2nd and 3rd and between those of the 6th and 7th variant in the ranking, shown in Table 8.3, need to be explored in more detail.

2nd in ranking: Variant SLC on PWD, CTC on MiniCTD, 72 Ah			
69.6 %			
-			
80	*120*	*60*	*40*
performance	**costs**	**physics**	**modifiability**
98.2 %	35 %	90.4 %	85.0 %
-	€136	-	-

3rd in ranking: Variant SLC on PWD, CTC on BCD, 61 Ah			
67.3 %			
-			
80	*120*	*60*	*40*
performance	**costs**	**physics**	**modifiability**
94.5 %	48 %	89.6 %	37.5 %
-	€124	-	-

6th in ranking: Variant SLC on BCD, CTC on BCD, 61 Ah			
60.1 %			
-			
80	*120*	*60*	*40*
performance	**costs**	**physics**	**modifiability**
60.2 %	64 %	92.3 %	0.0 %
-	€113	-	-

7th in ranking: Variant SLC on BCD, CTC on MiniCTD, 72 Ah			
59.5 %			
-			
80	*120*	*60*	*40*
performance	**costs**	**physics**	**modifiability**
65.5 %	44 %	90.4 %	47.5 %
-	€127	-	-

Table 8.3: Results of the 2nd, 3rd, 6th, and 7th variants in the ranking

The better results of Group A are mainly based on better modifiability and performance results. But, the difference of the modifiability between the 2nd and 3rd variant are the same as between the 6th and 7th variant. As modifiability evaluation is based on expert knowledge, its sensitivity—actually its architectural impact—can hardly be investigated without additional expert consultation. An expert, however, needs detailed information on the current situation in order to provide additional advice. The most influential quality attribute for the respective architecture variants seems to be costs. The cost difference between the 2nd and 3rd variant is €12 with a result difference of 13 percentage points. The cost difference between the 6th and 7th variant is €14 with a result difference of 20 percentage points. Obviously, the architectural impact of the changes between the respective variants changes as

well, i.e. from 1.08 % per € to 1.43 % per €. This can be interpreted as follows. Modifiability in terms of free resources causes costs. Because of additional resources provided by the architecture variants of Group A, not only the modifiability is raised but rather the performance. Higher performance is the reason for Group A being ranked higher than Group B despite the higher costs. But the additional expenses pay of twice, for performance and for modifiability. Additional modifiability can be achieved with less effort for Group A variants than for Group B ones. The lower costs difference in combination with lower costs sensitivity makes investments for modifiability more valuable. The more expensive and more modifiable architecture variants are preferred in Group A while the less expensive ones are preferred in Group B. Even a battery with more capacity for achieving a small physics result improvement is affordable in Group A.

Groups C and D The K.O. of the 50 Ah battery architecture variants with promising quality rates motivates the consideration of additional battery capacities. Actually, the architecture variants miss the K.O. limit by less than one day. With additional capacity, at least the architecture variant with 63.4 % quality may reach higher ranks. The standby time and life time attributes have 6.4 % structural impact each (see Section 5.1), which denotes the limit of percentage points contributing the overall evaluation result. Higher capacity of the battery will raise both results, which may be enough to push the architecture variant to the top. As other battery types are not available at the moment, estimations for new types have to be made or concrete offers need to be invited. However, when the information is available, the evaluation of new variants can be cost-efficiently performed because only costs and physics need to be re-evaluated. Expensive evaluations, e.g. performance and modifiability, can be reused as they are not effected by the battery capacity (see Section 6.1). To get an idea of the dimensioning of possible battery types, the 61 Ah battery can be taken as an upper limit. The evaluation results of the matching architecture variants with the 61 Ah batteries can even be used to estimate the upper limit of improvement by building new architecture variants.

Group E The reverse order of the battery capacity for the architecture variants with Reduced Convertible Top Device is noticeable because the 61 Ah battery architecture variants usually are preferred over those with 72 Ah capacity. Because of the K.O. result regarding the costs attribute, additional costs for a bigger battery do not matter anymore but the positive effect still exists. However, the difference between the results for 61 Ah architecture variants and those 72 Ah is comparatively small. Taking the results with 61 Ah battery capacity into account reveals that they already meet the requirements quite well (86 % for the battery attributes) and save some weight with respect to the bigger battery architecture variants. Again, this approves the 61 Ah battery as upper limit and guides future development in the opposite direction, i.e. the use of batteries with less capacity.

8.2 In-Car Radio Navigation System Analysis

For the In-Car Radio Navigation System case study, the tradeoff between performance and costs has been identified as relevant. The legitimate question arises how to correlate performance and costs without known correlation (see Figure 8.1). In case of costs, the interpretation shown in Figure 6.11 is based on the raw value of the evaluation result which is provided in €. Performance in terms of user noticeable quality is expressed in system reaction delay. Because of various use cases—in this case rather meaning user interactions—the interpretation has to be given with respect to the particular timing requirements of a particular use case. Figure 6.12 depicts the common interpretation of delay based on the expectations of the user. The tolerance limit, which represents to 100 % of the timing requirements, is applied as point of reference for all use cases.

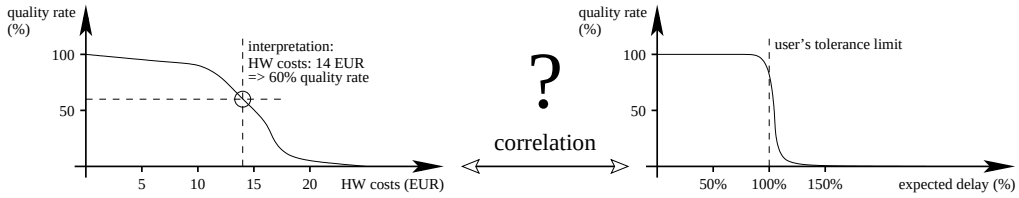

Figure 8.1: ICRNS interpretations to be correlated

8.2.1 In-Car Radio Navigation System Tradeoff Analysis

Although the existence of a costs performance tradeoff is well known, information on how to correlate those quality interpretations is quite rare. Especially a relation between expected delay with respect to various use cases and costs is not available without further analysis. Hence, performance analysis is just a first step of getting a link in a chain of dependencies needed for correlation. Furthermore, the performance analysis results themselves determine additional dependencies to be investigated. The MPA provides results describing dependencies in terms of delay of particular use cases in milliseconds over processor speed. While delays regarding various use cases can easily be aligned at the user's tolerance limit as stated above, the dependency of processor speed on costs still needs to be investigated. Actually, this can be done by inquiring a business department and sifting through some price lists. In this section, the steps to correlate the quality interpretations via a chain of dependencies are performed on the In-Car Radio Navigation System case study.

The evaluation is based on costs, performance, and modifiability as shown in Table 8.4. While performance is sufficient in all variants, costs reduction is the center of interest. But, costs and performance quality attributes are reciprocally influenced by changes in the architecture. MPA provides analysis results for the favorite Variant II as shown in Figure 7.10. The delay will grow if processor speed is decreased.

Weights and Impact										
25 %			50 %				25 %			
50			*100*				*50*			
costs			performance				modifiability			
12.5 %		12.5 %	33.33 %			16.67 %	12.5 %	12.5 %		
100		*100*	*200*			*100*	*100*	*100*		
HW costs		SW costs	devices			com	sav.pot.	scalab.		
6.25 %	6.25 %		8.33 %		25 %					
100	*100*		*50*		*150*					
devices	com		mem		CPU					
			4.17 %	4.17 %					impact	
			100	*100*					*weights*	
			RAM	ROM					QA name	

Table 8.4: ICRNS weights and impact

More delay means worse interpretation, i.e. a lower quality rate. In general, higher performance of a CPU is the equivalent to higher costs. Thus, in case of higher performance, the results of the cost quality attribute becomes worse. These facts are combined to uncover architecture potential (cf. Florentz [Flo07b, FH07]). Figures 8.2 to 8.7 depict input, intermediate steps, and output of the analysis for architecture potential. Following, the rationale and meaning of the coordinate systems is explained in detail.

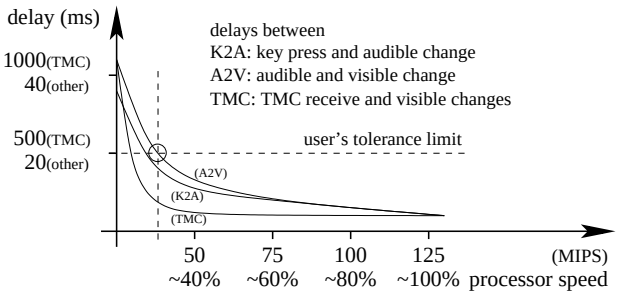

Figure 8.2: MPA results of MMI/RAD processor speed (Variant II)

Figure 8.2: MPA results of MMI/RAD processor speed (Variant II) This figure contains the MPA result for architecture Variant II (cf. Figures 7.9 and 7.10). It represents the dependencies between the MMI/RAD processor speed and the delays of some scenarios, i.e. use cases. The dependencies are nearly uniform, which is true for most delays observed by MPA in the case study. Thus, this result is taken as representative to save analysis effort. For more precise results, this analysis has to be performed for each of the architecture variants, each of the scenarios, and each of the processors deployed in a variant.

Additionally, the user's tolerance limit is given explicitly in the coordinate system. This is an important piece of information in order to be able to map costs to delay

via the processor speed. Because the quality rate interpretation for delay is given as percentage of expected delay and the dependency on processor speed is given as absolute values in milliseconds. Please note that the requirements taken from Wandeler et al. [WTVL06] are raised to design the analysis more expressive.

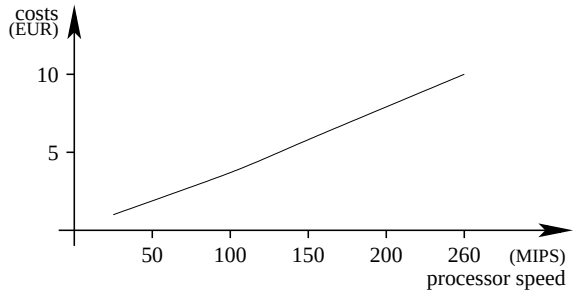

Figure 8.3: Costs dependency on processor speed

Figure 8.3: Costs for processor speed The dependency of processor speed on costs is depicted in this system. Such information has to be obtained from the business/purchasing department and may change over time. Thus, it is quite important to document such information for later reconstruction of architectural decisions.

This dependency is necessary in Architecture Potential Analysis because the costs interpretation needs to be correlated with the performance interpretation via the MPA results. Hence, a dependency between costs and the MPA results has to be identified. While the delay will be correlated with respect to the user's tolerance limit, the processor speed needs to be put into relation with costs. This is the rationale of the costs performance tradeoff. If there was no dependency between costs and processor speed, i.e. the hardware performance, this tradeoff would not exist.

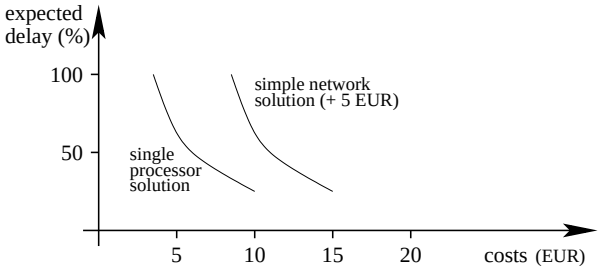

Figure 8.4: Costs-delay dependencies, left: without network, right: with network

Figure 8.4: Costs-delay dependencies left: without network, right: with network The dependency between costs and delay can be derived via the processor speed because the dependency *of* processor speed on costs and the dependency of

delay *on* processor speed are already known. The graphs show the costs of the hardware (without and with network) that keep the delay in the scope of the user's expectations. Until now, only processor costs have been taken into account as hardware costs. Therefore, a second dependency containing the network costs is shown in the coordinate system. This is necessary, because the interpretation of costs takes network costs into account. Although they are invariant regarding changes of the processor performance, they are not to be neglected. Thus, the additional dependency is given.

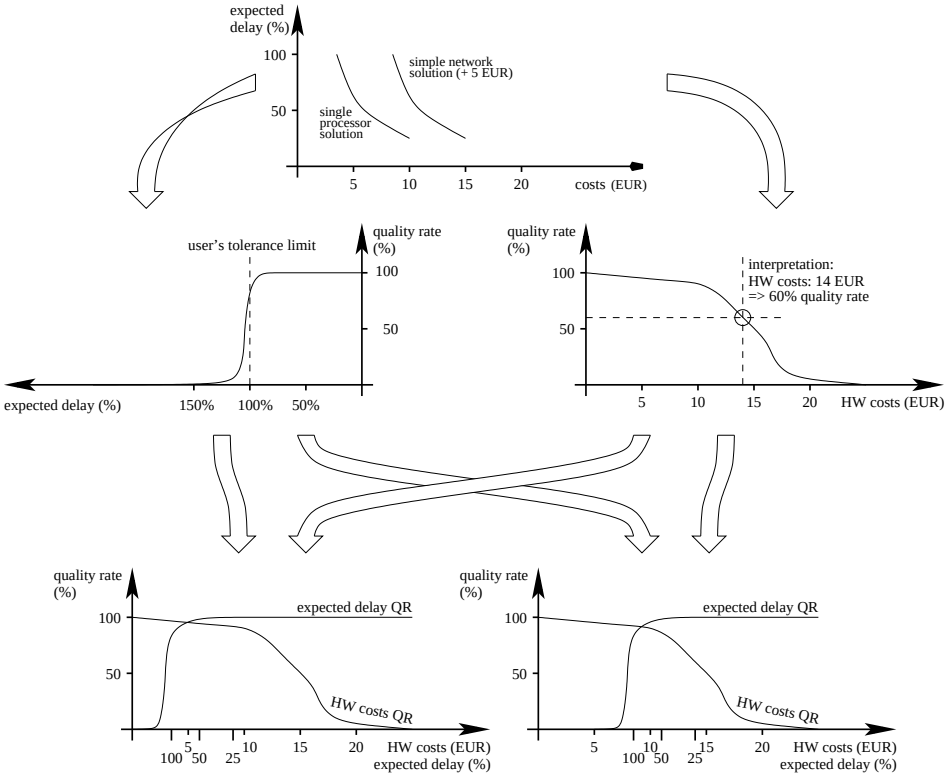

Figure 8.5: Correlation of hardware costs and (expected) delay

Figure 8.5: Correlation of hardware costs and (expected) delay The dependencies represented in Figure 8.4 are necessary to correlate hardware costs and (expected) delay as domains of the coordinate systems in Figures 8.6 and 8.7. In those coordinate systems, the architecture potential will be depicted, which to uncover and express was the main intention of the analysis. The reciprocal dependency between hardware costs and expected delay requires a change of the domain orientation of one of the interpretations for correlation. Without loss of generality, the expected delay interpretation has been selected for this change. Because of the network costs to be taken into account for network based architecture variants, two correlation need to be

provided. The bottom left coordinate system of Figure 8.5 contains the correlation not taking network costs into account. The lower right one contains network costs.

Figure 8.6 and 8.7: Architecture potential based on costs-delay dependency Both systems contain the same type of analysis result. Because of differences in Variants II and III (with and without network), the analysis results have to be presented separately. Thus, the position of the expected delay quality rate is shifted to the right with respect to the hardware costs quality rate in the second system (Variant II). This is necessary to regard network costs. The third graph in both systems represents the partial quality result based on the weighted summation of both quality rates. The distance to the maximum of this graph (see the arrow) presents the architecture potential for the variants. The results are discussed in detail below.

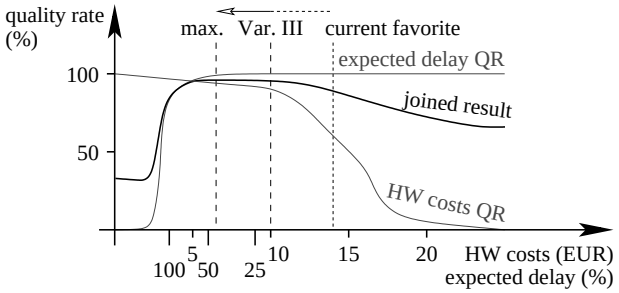

Figure 8.6: Architecture potential without network (Variant III)

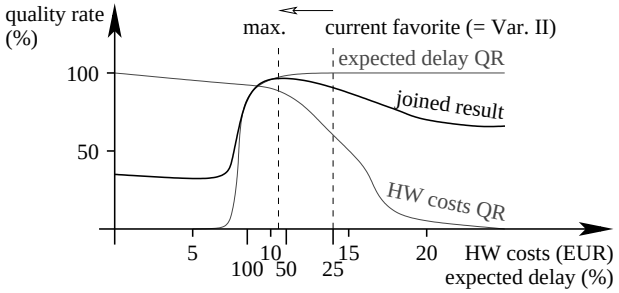

Figure 8.7: Architecture potential with network (Variant II)

8.2.2 In-Car Radio Navigation System Insights

The analysis of architecture potential uncovers not just potential regarding particular quality attributes. The documentation of the analysis results helps to express architectural knowledge.

First, the costs performance tradeoff can be considered based on the complete sensitivities instead of just sensitive points. Hence, not just the tradeoff itself but the impact of an architectural change is known. It can be predicted quite precisely by taking the intensity of the change into account. Furthermore, the tradeoff can be discussed regarding particular architecture variants for which its effects can be quite different.

Second, an extended view on the architecture rationale is provided. In the coordinate systems in Figures 8.6 and 8.7 is shown that the overall architecture potential regarding saving expenses is significantly lower for the network based solution. The single controller solution provides more potential on this score. The reason for this are the costs for the network, which are not considered in a processor down-scaling of the currently available architecture variants. Actually, these costs push the modifiability result because the system can be modified during design time and even life time without directly affecting the controllers. Thus, a low-budget version without navigation system can easily be realized. Moreover, the system can be offered with an optional navigation system (which can be refit later).

While more architecture potential is predicted for the single processor solution, the network solution already made it in the evaluation. To use the identified architecture potential, further considerations should be done. The identified potential is directed to save expenses while keeping the performance up. Modifiability is not yet included. Thus, the predicted improvements by downsizing one controller are carried forward to the evaluation to consider modifiability. Actually, this procedure is much simpler than extending the analysis for architecture potential. If the feedback reveals ambiguous results, the analysis can still be extended.

The result of a changed Variant II will be raised to approximately 97.8 %. The result of an even more changed Variant III will be raised to approximately 86.7 %. This seems to be surprising at first because Variant III may be less expensive. But, the original evaluation favored Variant II already because of its higher modifiability which has not been considered in the analysis for architecture potential. Furthermore, device costs have significantly less structural impact than the CPU performance in this case study (6.25 % vs. 25.0 %, see Table 8.4) and the architectural impact of costs is limited for Variant III as exemplarily depicted in Figure 5.3. Thus, although Variant III has more potential, this is not sufficient to make up the advantage of Variant II.

The nearly optimal fulfillment of the requirements by a changed Variant II do not motivate building additional architecture variants or making further attempts of improvement. At most down-scaling of the remaining controller of Variant II may be taken into account in this special case to achieve even more improvement. Variant III will not be regarded any more.

In a less definite case study, to know about the challenge of the evaluation (see Section 5.1) could be helpful to determine whether further attempts are promising.

9 Conclusion and Outlook

The summary of this work with respect to the objectives stated in Section 1.4 is presented in this chapter. Moreover, the adaptability of architecture evaluation—as contained in this work—to other application domains than the automotive area is addressed. Interesting fields for further research are discussed at the end of this chapter.

9.1 Conclusion

The main objective of this work is to provide decision support for architecture development to improve the development process. The decision support is mainly based on the transparency of architectural decisions provided for documentation, communication, and after all for reuse of appropriate decisions.

The metamodel called QADAG builds the backbone of the presented approach. The explicit representation of the architecture quality requirements is the starting point for understanding the rationale of architecture development. This again is the basis for communicating and justifying architectural decisions. For selection-based architecture evaluation, decisions regarding architecture variants to be kept in the development process are supported. The structured requirements representation using the QADAG provides the basis for efficient processing of the evaluation in order to identify promising variants and rejecting insufficient ones as early as possible. This helps to keep evaluation effort low which is quite short. Moreover, Architecture Potential Analysis can be performed to handle tradeoffs regarding competing requirements. For a selection of architecture variants, decisions regarding promising changes can be supported and risky ones can be avoided. Hence, architecture development can be integrated with architecture space exploration. This is a first step to manage the complexity of the solution space of possible architecture variants. Furthermore, changing approved architecture variants is one possibility of architectural decision reuse.

All in all, the architecture development process is improved as not only decision support can be provided but rather the documentation of architecture and architectural decisions is significantly improved. The communication of architecture rationals will become easier and even knowledge on architecture development can be passed on at an understandable level. Misleading architecture concepts can be avoided in future development, and approved ones can be reused with respect to the actual architecture quality requirements. The support of making, communicating, and reusing

architecture decisions is a key to improve the efficiency and reliability of architecture development. The idea of digital libraries of architecture components is picked up and carried on by this approach. With the availability of decision support, a tool-supported and tool-guided development becomes more and more conceivable. Not only the modeling but rather decision support will become the focus of development tools.

This approach is based on the quantification of architecture quality. Not quantifiable requirements and results can be integrated but limit the advantages. The overall change from qualitative requirements and results in architecture evaluation to quantitative ones can be performed without actual break in already applied development processes. Thus, a parallel processing of old standards and new methodology is not necessary as an integration of qualitative requirements in the quantitative approach is possible. Additional development effort required by this approach in terms of building QADAG instances and providing architecture details will pay off after a short time and furthermore will decline in future development projects. However, with the increased flexibility in architecture development as one goal of AUTOSAR, additional modeling effort will become necessary anyway to make use of the advantages aimed at by function-oriented development. After all, the change from controller-oriented and integration-based processes to function-oriented and architecture-based development becomes necessary to handle the growing complexity of software-intensive embedded systems in the automotive domain.

9.2 Adaptability

The adaptability of the approach presented in this work has first been mentioned in Section 2.2 with respect to the C&C viewtype. A more detailed consideration is given in the current section.

The underlying metamodel for views on embedded system architecture is capable of being adapted for use in other network-based embedded system domains. However, a flexible distribution of the functions to the hardware should still be intended in order to motivate the consideration of architecture in the particular domain. Besides adaptions of the architecture views, domain-specific quality attributes need to be defined to be applied in QADAG instances. However, the principles of evaluation and analysis stay the same as in the automotive domain. Hence, the approach presented in this work can be applied in other network-based embedded system domains without extensive adaption.

Embedded domains regarding systems not based on a controller network are interesting to be evaluated and analyzed as well. Although their variety is less distinctive than in network-based domains, architecture development is already part of the development processes in such domains. The underlying metamodel for architecture views needs to be exchanged in order to meet the specifics of the respective do-

main. Nevertheless, the QADAG as structured quality requirement representation can be applied without adaption except for the application of domain-specific quality attributes and thus evaluation techniques. The strong relation to hardware of embedded systems and short hardware resources still build a problem although the domains are not restricted to networks. Thus, the quantified representation of architecture requirements as well as of evaluation results bears the same benefits as in the automotive embedded domain. The explicit tradeoff consideration based on Architecture Potential Analysis stays applicable.

The adaptation to pure software architecture is more problematic. The relation to the underlying hardware is less concrete than in embedded systems domains. Following, the requirements usually are expressed more abstract. Again, the evaluation techniques build the interface between quality requirements and architecture model. Because of the more abstract requirement description, evaluation as well as analysis in general are more difficult and even less expressive. To overcome this lack of expressiveness, a reference realization platform can be used. Besides technical limits, an idea of the available hardware during architecture development is provided. The latter one again raises the understandability and traceability of the architectural decisions. Even without a concrete reference platform, a structured representation of architecture requirements and evaluation results in whichever expressiveness is not just nice to have but is an essential basis for efficient and successful development.

9.3 Outlook

A first step to establish this approach is an integration with tool support for architecture modeling. Although even manual handling of this approach is possible, for application in modern development processes, tool support is highly recommended. Another point of interest is the complexity based on the number of possible architecture variants. To decrease the overall development effort, this problem should be seriously taken into account. The estimation of the evaluation challenge as reference point for the evaluation results is promising in terms of motivating Architecture Potential Analysis and further architecture development. Even for automated development and optimization, knowledge of the challenge is quite useful.

Integration with Tool Support

Although managing the QADAG is no problem which requires tool support, the information contained in an instance of the QADAG needs to be documented in order to communicate the architecture quality requirements and architecture rationale. Especially for understanding past architectural decisions, knowledge of the architecture quality attributes, which motivated respective decisions, is essential. Moreover, decision support can be provided in terms of a guided development process based on

the exemplary representation of partial evaluation results with respect to responsible architecture parts or elements. However, the main advantage is directed to architecture modeling as basis for evaluation as well as for data acquisition according to the needs of the quality attributes.

Complexity based on the Number of Architecture Variants

Although the evaluation effort based on the amount of possible architecture variants for software-intensive embedded systems in the automotive domain has been addresses by the evaluation processing methodology in Chapter 5, much work needs to be done on complexity handling. The 18 variants of the Body Comfort System case study are based on only three decisions taking only strictly limited sets of alternatives on a small part of an embedded system into account. For complete systems, several hundreds of thousands of variants might be built. If a system is developed from scratch, i.e. no legacy systems are used as reference, the number of variants will even increase. As a strong selection is intended by the evaluation processing methodology of Chapter 5, feedback of evaluation results is one motivation of architecture analysis. However, for systems built from scratch, only few insights regarding the system quality are available. A structured variant building process is needed to guide the development including partial and iterative evaluation to identify promising decision variants and compose architecture variants covering different combinations of these decisions. As interdependencies are not yet considered by this decision-oriented procedure, it is less meaningful than architecture analysis. However, as preselection of architecture variants, it is one possibility worthwhile to be investigated for future developments.

Estimating the Challenge

Based on the analysis results and the experiences during system development, the evaluation challenge can be estimated as reference for the evaluation results. The more inner dependencies are known, the better the challenge estimation will be. With a decision-based procedure mentioned above, the challenge may even be assessable in early development phases in which few details on inner dependencies are known. Again, the consideration of legacy systems or at least parts of them may help to get information on the evaluation challenge. With approaches like AUTOSAR entering and even defining the development process, reuse of available and approved components and architectures will become common. However, as software-intensive embedded systems are in the center of interest for realizing innovations, new parts on which no initial quality statement can be made will be ubiquitous in the development of such systems. Thus, the evaluation challenge stays an interesting field of research especially in reuse-oriented development processes in the future.

Automated Development and Optimization

One of the main aspirations of tool supported architecture development is the automation of developing promising architecture variants from scratch as well as the automated optimization of such architecture variants. In most cases, architecture evaluation and analysis contains expert knowledge and integrates design space exploration. The latter usually is quite complex on its own. With growing experiences and availability of more detailed modeling of components and architecture, at least a wizard-like support for development becomes quite conceivable. Most of the dependencies needed for Architecture Potential Analysis are not known initially. The complexity of the architecture variants and even more of the dependencies between architecture and its quality impedes automated optimization. The identification and selection of relevant dependencies has been used in this approach to overcome this problem. A fully automated development and optimization process without expert interaction is not likely to be realized because of the complexity of architecture models and dependencies. Integrating decision support as presented in this approach with expert knowledge and interaction as well as with common development processes is more valuable at the moment. Moreover, fully automated development decreases experience effects as the results are rarely questioned. A feedback of experiences and export knowledge to automation can not be supported without expert interaction. Experts are needed to be integrated in the development process anyway for interaction during development or for setting up and running an automated process. Results have to be interpreted and analyzed as well, which again requires expert interaction. Thus, automation in terms of becoming independent from experts is not worthwhile. Automation and optimization should clearly be directed to support the handling of evaluation complexity and thus the development process itself.

Bibliography

[AAG95] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995. 4

[ABC+96] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie Mellon University, 1996. 63, 64

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. 138

[Bas93] Victor R. Basili. Applying the Goal/Question/Metric Paradigm in the Experience Factory. In *Proceedings of the Tenth Annual CSR (Centre for Software Reliability) Workshop, Application of Software Metrics and Quality Assurance in Industry*, September 1993. 79

[BBK03] Felix Bachmann, Len Bass, and Mark Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report CMU/SEI-2003-TR-004, Software Engineering Institute, Carnegie Mellon University, March 2003. 95

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998. xv, 3, 58, 59, 60, 67, 68, 69, 72, 81, 83, 103, 104, 132, 135, 144

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*, pages 528–532. John Wiley and Sons, 1994. 79

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems, 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT 2004)*, number 3185 in Lecture Notes of Computer Science, pages 200–236. Springer, 2004. 138

[BG04] Muhammad Ali Babar and Ian Gorton. Comparison of Scenario-Based Software Architecture Evaluation Methods. In *APSEC*, pages 600–607, 2004. 100

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture - A Pattern Language for Distributed Computing*, volume 4. Wiley & Sons, 2007. 7

[BK04] Gianluca Bontempi and Wido Kruijtzer. The use of intelligent data analysis techniques for system-level design: a software estimation example. *Soft Comput.*, 8(7):477–490, 2004. 8

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*, volume 1. Wiley & Sons, 1996. 7

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 87

[Bos06] Jan Bosch. Expanding the Scope of Software Product Families: Problems and Alternative Approaches. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Quality of Software Architectures (QoSA'06)*, volume 4214 of *Lecture Notes in Computer Science*, page 1. Springer, 2006. 87

[BRST05] Klaus Bergner, Andreas Rausch, Marc Sihling, and Thomas Ternité. DoSAM - Domain-Specific Software Architecture Comparison Model. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *Quality of Software Architectures and Software Quality*, volume 3712 of *Lecture Notes in Computer Science*, pages 4–20. Springer, September 2005. xvi, 79, 80

[BZJ04] Muhammad Ali Babar, Liming Zhu, and D. Ross Jeffery. A Framework for Classifying and Comparing Software Architecture Evaluation Methods. In *Australian Software Engineering Conference*, pages 309–319, 2004. 100

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002. xv, 1, 5, 6, 9, 13, 14, 15, 18, 19, 34

[CBKA95] Paul Clements, Len Bass, Rick Kazman, and Gregory Abowd. Predicting Software Quality by Architecture-Level Evaluation. In *Fifth International Conference on Software Quality*, Austin, TX, October 1995. 101

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. 139

[CKK01] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures. Methods and Case Studies.* Addison-Wesley, 2001. 86, 144

[Dav03] Alan M. Davis. System Phenotypes. *IEEE Software*, 20(4):54–56, 2003. 58

[DdHT01] Eric M. Dashofy, Andrew Van der Hoek, and Richard N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society. 5

[DN02] Liliana Dobrica and Eila Niemelä. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002. 100

[DSLT04] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST-ADL an Architecture Description Language, Validation and Verification Aspects. In *Workshop on Architecture Description Languages*, Toulouse, France, August 2004. 5

[ES05] Leire Etxeberria and Goiuria Sagardui. Product-Line Architecture: New Issues for Evaluation. In *SPLC*, pages 174–185, 2005. 87

[FGB+07] Bastian Florentz, Ursula Goltz, Jörn Chr. Braam, Rolf Ernst, and Thomas Saul. Architekturevaluation: Qualitätssicherung in frühen Entwicklungsphasen. In *Achtes Symposium AAET 2007 - Automatisierungs-, Assistenzsysteme und eingebettete Systeme für Transportmittel*, February 2007. 150

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction, February 2006. Technical Note. 5

[FH04] Bastian Florentz and Peter M. Hofmann. Ein Model Driven Architecture Ansatz für die Softwareentwicklung im Automotive-Bereich. In *24. Tagung 'Elektronik im Kfz' Haus der Technik*, Essen, Germany, Juny 2004. 1

[FH06] Bastian Florentz and Michaela Huhn. Embedded Systems Architecture: Evaluation and Analysis. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Quality of Software Architectures (QoSA'06)*, volume

4214 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2006. 19, 57, 61

[FH07] Bastian Florentz and Michaela Huhn. Architecture Potential Analysis: A Closer Look inside Architecture Evaluation. *Journal of Software (JSW)*, 2(4), 2007. to appear. 81, 84, 160

[FHB06] Bastian Florentz, Mirko Harms, and Andreas Breuer. Sicherstellung der Diagnosequalität im Automobilbereich durch modellbasierten Entwicklung. In *9. Fachtagung 'Entwurf komplexer Automatisierungssysteme' (EKA), Braunschweig,*, Braunschweig, Germany, May 2006. 1

[Flo06] Bastian Florentz. *Systemarchitekturevaluation: Integration unterschiedlicher Kriterien*, chapter 2.1, pages 49–65. Haus der Technik Fachbuch. expert verlag, 2006. 80

[Flo07a] Bastian Florentz. *Architekturevaluation: Schnittstelle zur Systemoptimierung*, chapter 6. Haus der Technik Fachbuch. expert verlag, 2007. 61, 81

[Flo07b] Bastian Florentz. Inside Architecture Evaluation: Analysis and Representation of Optimization Potential. In *Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, Los Alamitos, CA, USA, 2007. IEEE Computer Society. 61, 80, 81, 142, 160

[FMH04] Bastian Florentz, Martin Mutz, and Michaela Huhn. Avoiding Unpredicted Behaviour of Large Scale Embedded Systems by Design and Application of Modelling Rules. In *SIVOES-MoDeVa, Workshop at the 15th IEEE Intern. Symposium of Software Reliability Engineering (ISSRE)*, pages 59–66, November 2004. 1

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. 62

[Gil88] Tom Gilb. *Principles of Software Engineering Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 1988. 58

[Gil05] Tom Gilb. *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage.* Butterworth-Heinemann Ltd, August 2005. 58

[Gor06] Ian Gorton. *Essential Software Architecture.* Springer, Berlin, Germany, 2006. 1

[GP94] David Garlan and Dewayne Perry. Software Architecture: Practice, Potential, and Pitfalls. In *Proceedings of the 16th international conference on Software engineering (ICSE '94)*, pages 363–364, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 3

[Gru07] Lars Grunske. Early quality prediction of component-based systems - A generic framework. *Journal of Systems and Software*, 80(5):678–686, 2007. 100

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company. 1

[GS94] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994. 1

[GSZ+05] R. Gemmerich, S. Semmelrodt, A. Zündorf, C. Reckord, J. Leohold, J. Trippler, L. Brabetz, D. Müller, U. Schrey, and H.-G. Weil. Ein ganzheitlicher Ansatz zur Generierung und Optimierung von Fahrzeugbordnetzen. In *VDI Berichte Nr. 1907 (12th International Conference and Exhibition Electronic Systems for Vehicles Baden-Baden)*, pages 597–608, October 2005. 30, 32

[HE07] Arne Hamann and Rolf Ernst. Efficient Priority Optimization in Complex Distributed Embedded Systems through Search Space Adaptation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, January 2007. 150

[HHJ+05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System Level Performance Analysis - the SymTA/S Approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005. 150

[HMD+04] Michaela Huhn, Martin Mutz, Karsten Diethers, Bastian Florentz, and Michael Daginnus. Applications of Static Analysis on UML Models in the Automotive Domain. In *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT '04)*, pages 161–172, Brunswick, Germany, December 2004. 1

[HMF04] Michaela Huhn, Martin Mutz, and Bastian Florentz. A Lightweigth Approach to Critical Embedded Systems Design using UML. In *3rd International Workshop on Critical Systems Development with UML*, pages 28–40, October 2004. 1

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 1

[Hoc99] Elke HochMüller. Towards the Proper Integration of Extra-Functional Requirements. *The Australian Journal of Information Systems*, 7:98–117, 1999. Special Issue - Requirements Engineering. 58

[HRE06] Arne Hamann, Razvan Racu, and Rolf Ernst. Formal Methods for Automotive Platform Analysis and Optimization. In *Proceedings of Future Trends in Automotive Electronics and Tool Integration Workshop (DATE Conference)*, Munich, Germany, March 2006. 150

[HSF+04] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Mate, Kenji Nishikawam, and Thomas Scharnhorst. AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Convergence 2004, International Congress on Transportation Electronics*, October 2004. 1, 13, 17, 28, 114

[HV06] Martijn Hendriks and Marcel H. G. Verhoef. Timed Automata Based Analysis of Embedded Systems Architectures. In *Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*. IEEE Press, 2006. xvi, 138, 139

[IHO02] Mugurel T. Ionita, Dieter K. Hammer, and Henk Obbink. Scenario-Based Software Architecture Evaluation Methods: An Overview. In *Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering*, Orlando, Florida, USA, May 2002. 100

[ISO91a] ISO/IEC International Organization for Standardization. International Standard ISO/IEC 9126: Information technology - Software product evaluation - Quality characteristics and guidelines for their use, December 1991. 1st edn. 59, 76

[ISO91b] ISO/IEC International Organization for Standardization. International Standard ISO/IEC 9126: Software engineering - Product quality - Part 1: Quality model, December 1991. 1st edn. 77

[ISO03a] ISO/IEC International Organization for Standardization. Technical Report ISO/IEC TR 9126: Software engineering - Product quality - Part 2: External metrics. Technical Report ISO/IEC TR 9126-2:2003(E), ISO/IEC International Organization for Standardization, July 2003. 1st edn. 77

[ISO03b] ISO/IEC International Organization for Standardization. Technical Report ISO/IEC TR 9126: Software engineering - Product quality - Part 3: Internal metrics. Technical Report ISO/IEC TR 9126-3:2003(E), ISO/IEC International Organization for Standardization, July 2003. 1st edn. 77

[ISO04] ISO/IEC International Organization for Standardization. Technical Report ISO/IEC TR 9126: Software engineering - Product quality - Part 4: Quality in use metrics. Technical Report ISO/IEC TR 9126-4:2004(E), ISO/IEC International Organization for Standardization, April 2004. 1st edn. 77

[JKC04] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*, 21(5):88–92, 2004. 77

[JRH+04] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 Glasklar*. Carl Hanser Verlag, München, 2004. 14

[KABC96] Rick Kazman, Gregory D. Abowd, Leonhard J. Bass, and Paul Clements. Scenario-Based Analysis of Software Architecture. *IEEE Software*, 13(6):47–55, 1996. 78, 101

[KAK01] Rick Kazman, Jai Asundi, and Mark H. Klein. Quantifying the Costs and Benefits of Architectural Decisions. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 297–306, Washington, DC, USA, 2001. IEEE Computer Society. xvi, 78, 81, 147, 148

[KAK02] Rick Kazman, Jai Asundi, and Mark H. Klein. Making Architecture Design Decisions: An Economic Approach. Technical Report CMU/SEI-2002-TR-035, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 2002. xvi, 78, 81, 147, 148, 149

[KBK+99] Rick Kazman, Mario Barbacci, Mark H. Klein, S. Jeromy Carrière, and Steven G. Woods. Experience with Performing Architecture Tradeoff Analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 54–63, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press. 77

[KBWA94] Rick Kazman, Leonhard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM: A Method for Analyzing the Properties of Software Architectures. In *International Conference on Software Engineering*, pages 81–90, 1994. 101

[KJ04] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture - Patterns for Resource Management*, volume 3. Wiley & Sons, 2004. 7

[KK98] Rick Kazman and Mark H. Klein. Performing Architecture Tradeoff Analysis. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 85–88, New York, NY, USA, 1998. ACM Press. 77, 101

[KKB+98] Rick Kazman, Mark H. Klein, Mario R. Barbacci, Tom A. Longstaff, Howard F. Lipson, and S. Jeromy Carriere. The Architecture Tradeoff Analysis Method. In *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 68–78, Monterey, CA, August 1998. 77, 100, 101

[KKC00] Rick Kazman, Mark H. Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 2000. xvi, 77, 78, 79, 100, 101

[Kru95] Philippe Kruchten. Architectural Blueprints–The "4+1" View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995. 1

[KT94] Phillipe Kruchten and Christopher J. Thompson. An Object-Oriented, Distributed Architecture for Large-Scale Ada Systems. In *TRI-Ada '94: Proceedings of the conference on TRI-Ada '94*, pages 262–271, New York, NY, USA, 1994. ACM Press. 4

[LIN06] LIN Consortium. LIN Specification Package Revision 2.1, November 2006. 46, 106

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, 1973. 106, 124

[Mie07] Torben N. Mielke. Architekturevaluation und Sensitivität im Software- und Systementwurf. Master's thesis, Institute for Programming and Reactive Systems, Technical University at Brunswick, 2007. 45, 70, 105

[MRW83] James A. McCall, Paul K. Richards, and Gene F. Walters. Factors in Software Quality. Technical Report RADC-TR-83-175, Volume II, Rome Air Development Center, July 1983. 79

[OMG03] OMG - Object Management Group. OMG Unified Modeling Language Specification Version 1.5, 2003. Version 1.5. 14

[OMG06] OMG - Object Management Group. OMG SysML Specification Version 1.0, 2006. Version 1.0. 5

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. 1

[PWT05] Simon Perathoner, Ernesto Wandeler, and Lothar Thiele. Timed Automata Templates for Distributed Embedded System Architectures. TIK Report 233, Swiss Federal Institute of Technology, Zurich, Switzerland, November 2005. 138

[Rau02] Andreas Rau. *Model-Based Development of Embedded Automotive Control Systems*. PhD thesis, Fakultät für Informatik der Eberhard-Karls-Universität Tübingen, 2002. 1

[RE06] Kai Richter and Rolf Ernst. Real-Time Analysis as a Quality Feature: Automotive Use-Cases and Applications. In *Proceedings of Embedded World Conference*, February 2006. 150

[RH06] Ralf Reussner and Wilhelm Hasselbring, editors. *Handbuch der Software-Architektur*. Dpunkt Verlag, March 2006. 1

[Rob91] Robert Bosch GmbH. CAN Specification 2.0, 1991. 47

[RSB07] Thomas Ringler, Martin Simons, and Reinhold Beck. Reifegradsteigerung durch methodischen Architekturentwurf mit dem E/E-Konzeptwerkzeug. In *13th International Conference and Exhibition Electronic Systems for Vehicles Baden-Baden*, October 2007. 32

[SA89] Thomas L. Saaty and Joyce M. Alexander. *Conflict Resolution: The Analytic Hierachy Process*. Praeger, 1989. 73, 74

[Saa94] Thomas L. Saaty. How To Make A Decision: The Analytic Hierarchy Process. *Interfaces*, 24(6):19–43, 1994. 72, 73

[SEI07] SEI - Carnegie Mellon Software Engineering Institute. Homepage of the Software Engineering Institute. http://www.sei.cmu.edu/, September 2007. 3

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 1

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-Based Development of Embedded Systems. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 298–312, London, UK, 2002. Springer-Verlag. 1

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, volume 2. Wiley & Sons, 2000. 7

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time Calculus for Scheduling Hard Real-time Systeme. In *Proceedings of the IEEE Internation Conference on Circuits and Systems*, 2000. 137

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1^{st} edition, August 1993. 69, 97

[TW04] Lothar Thiele and Ernesto Wandeler. *Performance Analysis of Embedded Systems*, chapter 15. CRC Press, 2004. 137

[WTVL04] Ernesto Wandeler, Lothar Thiele, Marcel H. G. Verhoef, and Paul Lieverse. System Architecture Evaluation Using Modular Performance Analysis - A Case Study. In *ISoLA*, Paphos, Cyprus, October 2004. 137, 138

[WTVL06] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006. xv, xvi, 45, 56, 68, 105, 123, 125, 126, 137, 138, 139, 140, 141, 145, 161

[Zim80] Hubert Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980. 36, 37

A BCS Component Properties

type	width	period	notes
pw_req_x	3 bit	x ms	direction + mode
childlock	1 bit	50 ms	active, inactive
pw_enabled	1 bit	50 ms	
wpos_x	8 bit	x ms	
clamp15	3 bit	100 ms	
top_status	2 bit	100 ms	opened, closed, unlocked
top_error	1 bit	100 ms	
top_msg	1 bit	100 ms	in instrument cluster
top_warning	1 bit	100 ms	at speed exceeding 5 km/h
vehicle_speed	8 bit	100 ms	
permanetly provided/required as the case arises			
paddle_x	3 bit	x ms	up to 5 states
switch	1 bit	50 ms	
latch_x	1 bit	x ms	
motordrive	3 bit	10 ms	direction + 4 speeds
ticks	1 bit	10 ms	
force	8 bit	10 ms	as current

Table A.1: BCS signal properties

type	ROM util	RAM util	CPU util
LIN master 10 ms	2500 byte	100 byte	0.16 MIPS
LIN master 5 ms	3000 byte	120 byte	0.32 MIPS
LIN slave	1000 byte	100 byte	0.16 MIPS
RF receive	9500 byte	150 byte	0.16 MIPS
flash	8000 byte	-	-
OS 16 bit	15000 byte	180 byte	0.64 MIPS
OS 8 bit	11264 byte	50 byte	0.13 MIPS

Table A.2: BCS properties of other software

type	ROM util	RAM util	CPU util	
	required signals		provided signals	
PCC	35840 byte	320 byte	2.80 MIPS	
	dd_dd_pw_paddle_50 fp_fp_pw_paddle_100 rl_rl_pw_paddle_100 rr_rr_pw_paddle_100 dd_fp_pw_paddle_50 dd_rl_pw_paddle_50 dd_rr_pw_paddle_50 all_pw_paddle_50 childlock_switch clamp15 top_status dd_wpos_50 fp_wpos_100 rl_wpos_100 rr_wpos_100	*CLC input/output omitted*	dd_pw_req_50 fp_pw_req_100 rl_pw_req_100 rr_pw_req_100 childlock pw_enabled	
PWC	3072 byte	50 byte	0.80 MIPS	
	pw_req_50/100 pw_enabled motor_ticks motor_force		window_motordrive wpos_50/100	
SLC	3072 byte	30 byte	0.08 MIPS	
	door_latch pw_req_50/100 wpos_50/100		pw_req_40/100	
CTC	12000 byte	200 byte	0.40 MIPS	
	fl_top_latch fr_top_latch rl_top_latch rr_top_latch vehicle_speed		top_status top_error top_msg top_warning	
CLC	... 5000 byte	120 byte	0.16 MIPS	...
misc	... 68520 byte	2420 byte	0.88 MIPS	...

Table A.3: BCS function properties

type	cost	weight	signals
sensors			provided
power window paddle switch (up to five states)	€3.00	8 g	paddle
power window paddle switch (for simultaneous movement)	€2.00	10 g	paddle
driver power window panel (containing several switches, one toggle for child lock)	€5.00	70 g	switch paddle paddle paddle paddle
hall	€0.50	2 g	tick
force (as current)	€0.50	3 g	force
door latch	€4.00	10 g	latch
front top latch	€2.00	10 g	latch
rear top latch	€2.00	10 g	latch
actuators			required
powerwindow motor	€4.00	320 g	movement

Table A.4: BCS sensor and actuator properties

type	cost	ROM	RAM	freq.	CPU	weight	stby cur
BCD	€27.00	256 kB	8 kB	40 MHz	8.0 MIPS	480 g	4.50 mA
PWD	€ 6.00	16 kB	512 B	8 MHz	1.6 MIPS	95 g	0.25 mA
PWD (ext)	€10.00	24 kB	512 B	8 MHz	1.6 MIPS	100 g	0.35 mA
MiniCTD	€13.00	custom	custom	custom	custom	200 g	0.02 mA
RedCTD	€40.00	32 kB	4 kB	8 MHz	1.6 MIPS	400 g	0.12 mA

Table A.5: BCS ECU properties

from	to	cross-sect.	length
cables for communication lines			
left door latch	BCD	0.70 mm^2	216 cm
right door latch	BCD	0.70 mm^2	250 cm
driver panel	PWD	3.50 mm^2	67 cm
front passenger paddle switch	PWD	1.05 mm^2	24 cm
left rear paddle switch	PWD	1.05 mm^2	19 cm
right rear paddle switch	PWD	1.05 mm^2	19 cm
left front top latch	CTD	0.70 mm^2	442 cm
right front top latch	CTD	0.70 mm^2	526 cm
left rear top latch	CTD	0.70 mm^2	108 cm
right rear top latch	CTD	0.70 mm^2	192 cm
left front top latch	BCD	0.70 mm^2	122 cm
right front top latch	BCD	0.70 mm^2	206 cm
left rear top latch	BCD	0.70 mm^2	449 cm
right rear top latch	BCD	0.70 mm^2	533 cm
additional paddle switch	BCD	1.05 mm^2	134 cm
additional paddle switch	PWD	1.05 mm^2	36 cm
additional paddle switch	CTD	1.05 mm^2	358 cm
left door latch	PWD	0.70 mm^2	163 cm
right door latch	PWD	0.70 mm^2	163 cm
cables for buses			
LIN, 19.2 kBits/s		0.35 mm^2	569 cm
low speed CAN, 100 kBits/s		0.70 mm^2	341 cm
CAN extension		0.70 mm^2	31 cm

Table A.6: BCS cable properties

capacity	cost	weight
50 Ah	€18	16 kg
61 Ah	€20	17 kg
71 Ah	€24	20 kg

Table A.7: BCS battery properties

from	to	period	width	signals
BCD	PWD (all)	50 ms	2 byte	rl_pw_req_100 (3 bit) rr_pw_req_100 (3 bit) pw_enable (1 bit) clamp15 (1 bit) *collapsed* 7 × 1 bit
BCD	PWD (front)	40 ms	2 byte	dd_pw_req_40 (3 bit) fp_pw_req_40 (3 bit) *collapsed* 3 × 1 bit
DD_PWD	BCD	50 ms	2 byte	dd_wpos_50 (8 bit) dd_dd_pw_paddle_50 (3 bit) *collapsed* 1 × 2 bit *collapsed* 2 × 1 bit
DD_PWD	BCD	50 ms	2 byte	dd_fp_pw_paddle_50 (3 bit) dd_rl_pw_paddle_50 (3 bit) dd_rr_pw_paddle_50 (3 bit) childlock_switch (1 bit) *collapsed* 2 × 1 bit
FP_PWD	BCD	100 ms	2 byte	fp_wpos_100 (8 bit) fp_fp_pw_paddle_100 (3 bit) *collapsed* 1 × 2 bit *collapsed* 3 × 1 bit
RL_PWD	BCD	100 ms	2 byte	rl_wpos_100 (8 bit) rr_rr_pw_paddle_100 (3 bit) *collapsed* 1 × 2 bit *collapsed* 3 × 1 bit
RR_PWD	BCD	100 ms	2 byte	rr_wpos_100 (8 bit) rr_rr_pw_paddle_100 (3 bit) *collapsed* 1 × 2 bit *collapsed* 3 × 1 bit
BCD	MFSW	100 ms	2 byte	*collapsed* 1 × 7 bit *collapsed* 4 × 1 bit
MFSW	BCD	50 ms	2 byte	*collapsed* 2 × 8 bit
MFSW	BCD	50 ms	2 byte	*collapsed* 1 × 16 bit
MFSW	BCD	100 ms	2 byte	*collapsed* 2 × 4 bit *collapsed* 8 × 1 bit
MFSW	BCD	50 ms	2 byte	*collapsed* 1 × 4 bit *collapsed* 11 × 1 bit

Table A.8: BCS LIN messages for 5 ms period schedule

from	to	period	width	signals
BCD	PWD (all)	50 ms	3 byte	dd_pw_req_50 (3 bit) fp_pw_req_100 (3 bit) rl_pw_req_100 (3 bit) rr_pw_req_100 (3 bit) clamp15 (1 bit) pw_enable (1 bit) *collapsed* 10 × 1 bit
DD_PWD	BCD	50 ms	4 byte	dd_wpos_50 (8 bit) dd_dd_pw_paddle_50 (3 bit) dd_fp_pw_paddle_100 (3 bit) dd_rl_pw_paddle_100 (3 bit) dd_rr_pw_paddle_100 (3 bit) childlock_switch (1 bit) *collapsed* 1 × 2 bit *collapsed* 4 × 1 bit
FP_PWD	BCD	100 ms	2 byte	fp_wpos_100 (8 bit) fp_fp_pw_paddle_100 (3 bit) *collapsed* 1 × 2 bit *collapsed* 3 × 1 bit
RL_PWD	BCD	100 ms	2 byte	rl_wpos_100 (8 bit) rl_rl_pw_paddle_100 (3 bit) *collapsed* 1 × 2 bit *collapsed* 3 × 1 bit
RR_PWD	BCD	100 ms	2 byte	rr_wpos_100 (8 bit) rr_rr_pw_paddle_100 (3 bit) *collapsed* 1 × 2 bit *collapsed* 3 × 1 bit
BCD	MFSW	100 ms	2 byte	*collapsed* 1 × 7 bit *collapsed* 4 × 1 bit
MFSW	BCD	50 ms	8 byte	*collapsed* 1 × 16 bit *collapsed* 2 × 8 bit *collapsed* 3 × 4 bit *collapsed* 20 × 1 bit

Table A.9: BCS LIN messages for 10 ms period schedule

message	period	width	signals
airbag (via gateway)	20 ms	4 byte	*collapsed*
confidential	800 ms	2 byte	*collapsed*
BCD1	500 ms	6 byte	*collapsed*
BCD2	200 ms	4 byte	*collapsed*
BCD3	50 ms	5 byte	*collapsed*
air-con	100 ms	8 byte	*collapsed*
diagnosis	1000 ms	8 byte	*collapsed*
confidential	200 ms	2 byte	*collapsed*
confidential	200 ms	4 byte	*collapsed*
confidential	1000 ms	2 byte	*collapsed*
confidential	200 ms	8 byte	*collapsed*
confidential	500 ms	3 byte	*collapsed*
confidential	20 ms	4 byte	*collapsed*
gear (via gateway)	100 ms	5 byte	*collapsed*
confidential	200 ms	5 byte	*collapsed*
engine (via gateway)	100 ms	3 byte	*collapsed*
confidential	100 ms	4 byte	*collapsed*
inst-cluster (via gateway)	100 ms	3 byte	*collapsed*
confidential	200 ms	2 byte	*collapsed*
confidential	100 ms	8 byte	*collapsed*
confidential	50 ms	4 byte	*collapsed*
confidential	1000 ms	4 byte	*collapsed*
confidential	100 ms	5 byte	*collapsed*
confidential	500 ms	3 byte	*collapsed*
confidential	500 ms	4 byte	*collapsed*
MFSW (via gateway)	100 ms	2 byte	*collapsed*
confidential	100 ms	8 byte	*collapsed*
PDC	500 ms	4 byte	*collapsed*
confidential	200 ms	3 byte	*collapsed*
confidential	500 ms	8 byte	*collapsed*
confidential	100 ms	2 byte	*collapsed*
confidential	200 ms	1 byte	*collapsed*
AHS	200 ms	4 byte	*collapsed*
confidential	500 ms	6 byte	*collapsed*
confidential	100 ms	1 byte	*collapsed*
CTC	100 ms	5 byte	top_status (2 bit) top_error (1 bit) top_msg (1 bit) top_warning (1 bit) *collapsed* 1 × 30 bit
confidential	200 ms	1 byte	*collapsed*
confidential	200 ms	2 byte	*collapsed*
confidential	100 ms	1 byte	*collapsed*
confidential	40 ms	5 byte	*collapsed*
confidential	20 ms	3 byte	*collapsed*
confidential	50 ms	1 byte	*collapsed*
confidential	200 ms	7 byte	*collapsed*

Table A.10: BCS CAN messages

B BCS Evaluation Results

The Body Comfort System case study has been evaluated in Section 6.1. Besides a detailed discussion of the evaluation and its results with respect to specific quality attributes, the tableviews of four of the architecture variants have been presented in that section. The remaining tableviews are presented in the current section. Table 6.6 on Page 121 contains an overview of the results including the location of the respective tables representing the results.

Variant SLC on PWD, CTC on MiniCTD, 72 Ah											
69.6 %											
-											
80			*120*			*60*			*40*		
performance			**costs**			**physics**			**modifiability**		
98.2 %			35 %			90.4 %			85.0 %		
-			€136			-			-		
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*	
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**	
98 %	98.3 %		-	-	-	76 %	100.0 %		100 %	70 %	
50 %	-		€92	€17	€27	22.6 kg	-		-	-	
	100	*100*	*100*				*100*	*100*			
	CPU	**RAM**	**ROM**				**stby**	**life**			
	100 %	100 %	95 %				100 %	100 %			
	60 %	68 %	73 %				57 d	4.59 a			

(weights)
QA name
quality rate
result

Table B.1: Evaluation results Variant SLC on PWD, CTC on MiniCTD, 72 Ah

Variant SLC on PWD, CTC on BCD, 72 Ah											
65.3 %											
-											
80			*120*			*60*			*40*		
performance			**costs**			**physics**			**modifiability**		
94.5 %			42 %			91.6 %			37.5 %		
-			€128			-			-		
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*	
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**	
98 %	91.0 %		-	-	-	79 %	100.0 %		75 %	0 %	
50 %	-		€84	€17	€27	22.4 kg	-		-	-	
	100	*100*	*100*				*100*	*100*			
	CPU	**RAM**	**ROM**				**stby**	**life**			
	98 %	100 %	75 %				100 %	100 %			
	65 %	70 %	77 %				57 d	4.59 a			

(weights)
QA name
quality rate
result

Table B.2: Evaluation results Variant SLC on PWD, CTC on BCD, 72 Ah

Variant SLC on BCD, CTC on MiniCTD, 61 Ah													
61.5 %													
-													
80			*120*				*60*			*40*			
performance			**costs**				**physics**			**modifiability**			
65.5 %			49 %				90.4 %			47.5 %			
-			€123				-			-			
100	*100*		*100*	*100*	*100*		*80*	*120*		*100*	*100*		
com	**ECU**		**ECU**	**s+a**	**e.sys**		**weight**	**batt**		**ext**	**scal**		
34 %	97.0 %		-	-	-		94 %	88.0 %		25 %	70 %		
65 %	-		€86	€13	€24		19.6 kg	-		-	-		
	100	*100*	*100*					*100*	*100*				
	CPU	**RAM**	**ROM**					**stby**	**life**				
	100 %	100 %	91 %					96 %	80 %				
	63 %	69 %	74 %					49 d	3.89 a				

(weights)
QA name
quality rate
result

Table B.3: Evaluation results Variant SLC on BCD, CTC on MiniCTD, 61 Ah

Variant SLC on BCD, CTC on MiniCTD, 72 Ah													
59.5 %													
-													
80			*120*				*60*			*40*			
performance			**costs**				**physics**			**modifiability**			
65.5 %			44 %				90.4 %			47.5 %			
-			€127				-			-			
100	*100*		*100*	*100*	*100*		*80*	*120*		*100*	*100*		
com	**ECU**		**ECU**	**s+a**	**e.sys**		**weight**	**batt**		**ext**	**scal**		
34 %	97.0 %		-	-	-		76 %	100.0 %		25 %	70 %		
65 %	-		€86	€13	€28		22.6 kg	-		-	-		
	100	*100*	*100*					*100*	*100*				
	CPU	**RAM**	**ROM**					**stby**	**life**				
	100 %	100 %	91 %					100 %	100 %				
	63 %	69 %	74 %					58 d	4.6 a				

(weights)
QA name
quality rate
result

Table B.4: Evaluation results Variant SLC on BCD, CTC on MiniCTD, 72 Ah

Variant SLC on BCD, CTC on BCD, 72 Ah													
58.0 %													
-													
80			*120*				*60*			*40*			
performance			**costs**				**physics**			**modifiability**			
60.2 %			59 %				91.6 %			0.0 %			
-			€117				-			-			
100	*100*		*100*	*100*	*100*		*80*	*120*		*100*	*100*		
com	**ECU**		**ECU**	**s+a**	**e.sys**		**weight**	**batt**		**ext**	**scal**		
34 %	86.3 %		-	-	-		79 %	100.0 %		0 %	0 %		
65 %	-		€78	€13	€26		22.3 kg	-		-	-		
	100	*100*	*100*					*100*	*100*				
	CPU	**RAM**	**ROM**					**stby**	**life**				
	95 %	100 %	64 %					100 %	100 %				
	68 %	71 %	79 %					58 d	4.60 a				

(weights)
QA name
quality rate
result

Table B.5: Evaluation results Variant SLC on BCD, CTC on BCD, 72 Ah

Variant SLC on BCD, CTC on MiniCTD, 50 Ah										
54.6 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
65.5 %			53 %			47.8 %			47.5 %	
-			€121			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
34 %	97.0 %		-	-	-	97 %	15.0 %		25 %	70 %
65 %	-		€86	€13	€22	18.6 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	91 %				1 %	29 %		
	63 %	69 %	74 %				40 d	3.19 a		

(weights)
QA name
quality rate
result

Table B.6: Evaluation results Variant SLC on BCD, CTC on MiniCTD, 50 Ah

Variant SLC on PWD, CTC on MiniCTD, 50 Ah										
63.4 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
98.2 %			41 %			47.5 %			85.0 %	
-			€130			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	98.3 %		-	-	-	97 %	14.5 %		100 %	70 %
50 %	-		€92	€17	€21	18.6 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	95 %				0 %	29 %		
	60 %	68 %	73 %				39 d	3.19 a		

(weights)
QA name
quality rate
result

Table B.7: Evaluation results Variant SLC on PWD, CTC on MiniCTD, 50 Ah

Variant SLC on PWD, CTC on BCD, 50 Ah										
60.2 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
94.5 %			51 %			47.9 %			37.5 %	
-			€122			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	91.0 %		-	-	-	98 %	14.5 %		75 %	0 %
50 %	-		€84	€17	€21	18.4 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	98 %	100 %	75 %				0 %	29 %		
	65 %	70 %	77 %				39 d	3.19 a		

(weights)
QA name
quality rate
result

Table B.8: Evaluation results Variant SLC on PWD, CTC on BCD, 50 Ah

Variant SLC on PWD, CTC on redCTD, 72 Ah										
56.5 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
98.2 %			0 %			89.2 %			90.0 %	
-			€167			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	98.3 %		-	-	-	73 %	100.0 %		100 %	80 %
50 %	-		€119	€17	€31	22.8 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	95 %				100 %	100 %		
	60 %	68 %	73 %				57 d	4.59 a		

(weights)
QA name
quality rate
result

Table B.9: Evaluation results Variant SLC on PWD, CTC on RedCTD, 72 Ah

Variant SLC on PWD, CTC on redCTD, 61 Ah										
55.9 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
98.2 %			0 %			88.8 %			90.0 %	
-			€163			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	99.0 %		-	-	-	93 %	86.0 %		100 %	80 %
50 %	-		€119	€17	€27	19.8 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	95 %				92 %	80 %		
	60 %	68 %	73 %				48 d	3.89 a		

(weights)
QA name
quality rate
result

Table B.10: Evaluation results Variant SLC on PWD, CTC on RedCTD, 61 Ah

Variant SLC on PWD, CTC on redCTD, 50 Ah										
47.6 %										
-										
80			*120*			*60*			*40*	
performance			**costs**			**physics**			**modifiability**	
98.5 %			0 %			47.1 %			90.0 %	
-			€161			-			-	
100	*100*		*100*	*100*	*100*	*80*	*120*		*100*	*100*
com	**ECU**		**ECU**	**s+a**	**e.sys**	**weight**	**batt**		**ext**	**scal**
98 %	98.3 %		-	-	-	96 %	14.5 %		100 %	80 %
50 %	-		€119	€17	€25	18.8 kg	-		-	-
	100	*100*	*100*				*100*	*100*		
	CPU	**RAM**	**ROM**				**stby**	**life**		
	100 %	100 %	95 %				0 %	29 %		
	60 %	68 %	73 %				39 d	3.19 a		

(weights)
QA name
quality rate
result

Table B.11: Evaluation results Variant SLC on PWD, CTC on RedCTD, 50 Ah

Variant SLC on BCD, CTC on redCTD, 72 Ah																
42.3 %																
-																
80					*120*				*60*				*40*			
performance					**costs**				**physics**				**modifiability**			
65.5 %					0 %				89.2 %				52.5 %			
-					€ 155				-				-			
100	*100*				*100*	*100*	*100*		*80*	*120*			*100*	*100*		
com	**ECU**				**ECU**	**s+a**	**e.sys**		**weight**	**batt**			**ext**	**scal**		
34 %	97.0 %				-	-	-		73 %	100.0 %			25 %	80 %		
65 %	-				€ 113	€ 13	€ 29		22.8 kg	-			-	-		
	100	*100*	*100*							*100*	*100*					
	CPU	**RAM**	**ROM**							**stby**	**life**		*(weights)*			
	100 %	100 %	91 %							100 %	100 %		**QA name**			
	63 %	69 %	74 %							57 d	4.60 a		quality rate			
													result			

Table B.12: Evaluation results Variant SLC on BCD, CTC on RedCTD, 72 Ah

Variant SLC on BCD, CTC on redCTD, 61 Ah																
42.2 %																
-																
80					*120*				*60*				*40*			
performance					**costs**				**physics**				**modifiability**			
65.5 %					0 %				88.8 %				52.5 %			
-					€ 151				-				-			
100	*100*				*100*	*100*	*100*		*80*	*120*			*100*	*100*		
com	**ECU**				**ECU**	**s+a**	**e.sys**		**weight**	**batt**			**ext**	**scal**		
34 %	97.0 %				-	-	-		93 %	86.0 %			25 %	80 %		
65 %	-				€ 113	€ 13	€ 25		19.8 kg	-			-	-		
	100	*100*	*100*							*100*	*100*					
	CPU	**RAM**	**ROM**							**stby**	**life**		*(weights)*			
	100 %	100 %	91 %							92 %	80 %		**QA name**			
	63 %	69 %	74 %							48 d	3.89 a		quality rate			
													result			

Table B.13: Evaluation results Variant SLC on BCD, CTC on RedCTD, 61 Ah

Variant SLC on BCD, CTC on redCTD, 50 Ah																
34.0 %																
-																
80					*120*				*60*				*40*			
performance					**costs**				**physics**				**modifiability**			
65.5 %					0 %				47.4 %				52.5 %			
-					€ 149				-				-			
100	*100*				*100*	*100*	*100*		*80*	*120*			*100*	*100*		
com	**ECU**				**ECU**	**s+a**	**e.sys**		**weight**	**batt**			**ext**	**scal**		
34 %	97.0 %				-	-	-		96 %	15.0 %			25 %	80 %		
65 %	-				€ 113	€ 13	€ 23		18.8 kg	-			-	-		
	100	*100*	*100*							*100*	*100*					
	CPU	**RAM**	**ROM**							**stby**	**life**		*(weights)*			
	100 %	100 %	91 %							1 %	29 %		**QA name**			
	63 %	69 %	74 %							40 d	3.19 a		quality rate			
													result			

Table B.14: Evaluation results Variant SLC on BCD, CTC on RedCTD, 50 Ah